PB3D

[2.47]

1 Welcome to PB3D 1

2 Input variables 3
2.1 PB3D Inputs . . . 3
2.2 POST Inputs . . . 15

3 Frequently Asked Questions 23
3.1 runtime errors . . . 23
3.2 common problems . . . 24

4 Code outputs 27
4.1 Output Files . . . 27
4.2 Output Plots . . . 30

5 General code structure 33
5.1 PB3D flowchart . . . 33
5.2 Equilibrium & Perturbation Jobs . . . 34
5.3 Detailed PB3D Flowchart . . . 35
5.4 POST Overview . . . 37

6 Tutorial 39
6.1 Getting the Equilibrium . . . 39
6.2 Checking the equilibrium (optional) . . . 40
6.3 Setting Up the Input File . . . 42
6.4 Running PB3D . . . 43
6.5 Run Script Tools . . . 45
6.6 Post-processing With POST . . . 45

7 Installation 47
7.1 Introduction . . . 47
7.2 Compilation . . . 49
7.3 Makefile Example . . . 49

8 Todo List 53

9 Directory Documentation 55
9.1 Documentation Directory Reference . . . 55
9.2 /Users/toonweyens/Code/PB3D/Modules Directory Reference . . . 55
9.3 /Users/toonweyens/Code/PB3D Directory Reference . . . 56

Appendices

Appendix A Programs 59
A.1 /Users/toonweyens/Code/PB3D/POST.f90 Program Rerefence . . . 59
A.2 /Users/toonweyens/Code/PB3D/PB3D.f90 Program Rerefence . . . 61

Appendix B Modules 63
B.1 driver_eq Module Reference . . . 63
B.2 driver_post Module Reference . . . 66
B.3 driver_sol Module Reference . . . 71
B.4 driver_x Module Reference . . . 76
B.5 dtorh Module Reference . . . 78
B.6 eq_ops Module Reference . . . 81
B.7 eq_utilities Module Reference . . . 100
B.8 eq_vars Module Reference . . . 105
B.9 files_ops Module Reference . . . 113
B.10 files_utilities Module Reference . . . 118
B.11 grid_ops Module Reference . . . 123
B.12 grid_utilities Module Reference . . . 136
B.13 grid_vars Module Reference . . . 153
B.14 hdf5_ops Module Reference . . . 158
B.15 hdf5_utilities Module Reference . . . 174
B.16 hdf5_vars Module Reference . . . 178
B.17 helena_ops Module Reference . . . 182
B.18 helena_vars Module Reference . . . 188
B.19 input_ops Module Reference . . . 193
B.20 input_utilities Module Reference . . . 197
B.21 messages Module Reference . . . 202
B.22 mpi_ops Module Reference . . . 214
B.23 mpi_utilities Module Reference . . . 219
B.24 mpi_vars Module Reference . . . 227
B.25 num_ops Module Reference . . . 229
B.26 num_utilities Module Reference . . . 231
B.27 num_vars Module Reference . . . 249
B.28 output_ops Module Reference . . . 275
B.29 pb3d_ops Module Reference . . . 280
B.30 pb3d_utilities Module Reference . . . 294
B.31 rich_ops Module Reference . . . 297
B.32 rich_vars Module Reference . . . 304
B.33 slepc_ops Module Reference . . . 307
B.34 slepc_utilities Module Reference . . . 319
B.35 sol_ops Module Reference . . . 322
B.36 sol_utilities Module Reference . . . 331
B.37 sol_vars Module Reference . . . 335
B.38 str_utilities Module Reference . . . 337
B.39 test Module Reference . . . 348
B.40 vac_ops Module Reference . . . 350

B.41 vac_utilities Module Reference . . . 361
B.42 vac_vars Module Reference . . . 366
B.43 vmec_ops Module Reference . . . 371
B.44 vmec_utilities Module Reference . . . 373
B.45 vmec_vars Module Reference . . . 375
B.46 x_ops Module Reference . . . 381
B.47 x_utilities Module Reference . . . 394
B.48 x_vars Module Reference . . . 401

Appendix C Interfaces and Types **411**
C.1 num_utilities::add_arr_mult Interface Reference . . . 411
C.2 mpi_utilities::broadcast_var Interface Reference . . . 414
C.3 num_utilities::bubble_sort Interface Reference . . . 418
C.4 num_utilities::calc_det Interface Reference . . . 420
C.5 eq_ops::calc_eq Interface Reference . . . 421
C.6 grid_utilities::calc_eqd_grid Interface Reference . . . 424
C.7 eq_utilities::calc_f_derivs Interface Reference . . . 426
C.8 eq_ops::calc_g_c Interface Reference . . . 427
C.9 eq_ops::calc_g_f Interface Reference . . . 429
C.10 eq_ops::calc_g_h Interface Reference . . . 431
C.11 eq_ops::calc_g_v Interface Reference . . . 433
C.12 num_utilities::calc_int Interface Reference . . . 434
C.13 num_utilities::calc_inv Interface Reference . . . 437
C.14 eq_utilities::calc_inv_met Interface Reference . . . 438
C.15 eq_ops::calc_jac_f Interface Reference . . . 440
C.16 eq_ops::calc_jac_h Interface Reference . . . 442
C.17 eq_ops::calc_jac_v Interface Reference . . . 444
C.18 num_utilities::calc_mult Interface Reference . . . 445
C.19 eq_ops::calc_rzl Interface Reference . . . 448
C.20 eq_ops::calc_t_hf Interface Reference . . . 449
C.21 eq_ops::calc_t_vc Interface Reference . . . 451
C.22 eq_ops::calc_t_vf Interface Reference . . . 452
C.23 grid_utilities::calc_tor_diff Interface Reference . . . 454
C.24 x_ops::calc_x Interface Reference . . . 456
C.25 sol_utilities::calc_xuq Interface Reference . . . 458
C.26 num_ops::calc_zero_hh Interface Reference . . . 461
C.27 num_utilities::con Interface Reference . . . 464
C.28 num_utilities::con2dis Interface Reference . . . 468
C.29 pb3d_utilities::conv_1d2nd Interface Reference . . . 469
C.30 num_utilities::conv_mat Interface Reference . . . 472
C.31 grid_utilities::coord_e2f Interface Reference . . . 476
C.32 grid_utilities::coord_f2e Interface Reference . . . 478

C.33 hdf5_vars::dealloc_var_1d Interface Reference 480
C.34 hdf5_vars::dealloc_xml_str Interface Reference 481
C.35 num_utilities::dis2con Interface Reference 482
C.36 eq_vars::eq_1_type Type Reference 483
C.37 eq_vars::eq_2_type Type Reference 488
C.38 vmec_utilities::fourier2real Interface Reference 496
C.39 mpi_utilities::get_ghost_arr Interface Reference 498
C.40 mpi_utilities::get_ser_var Interface Reference 500
C.41 grid_vars::grid_type Type Reference 501
C.42 hdf5_vars::hdf5_file_type Type Reference 507
C.43 mpi_vars::lock_type Type Reference 508
C.44 x_vars::modes_type Type Reference 511
C.45 num_utilities::order_per_fun Interface Reference 512
C.46 output_ops::plot_hdf5 Interface Reference 514
C.47 output_ops::print_ex_2d Interface Reference 518
C.48 output_ops::print_ex_3d Interface Reference 521
C.49 eq_ops::print_output_eq Interface Reference 523
C.50 x_ops::print_output_x Interface Reference 525
C.51 eq_ops::redistribute_output_eq Interface Reference 528
C.52 x_ops::redistribute_output_x Interface Reference 530
C.53 hdf5_ops::reset_hdf5_item Interface Reference 531
C.54 num_utilities::round_with_tol Interface Reference 534
C.55 x_utilities::sec_ind_loc2tot Interface Reference 535
C.56 x_vars::set_nm_x Interface Reference 536
C.57 sol_vars::sol_type Type Reference 538
C.58 num_utilities::spline Interface Reference 542
C.59 eq_utilities::transf_deriv Interface Reference 545
C.60 vac_vars::vac_type Type Reference 549
C.61 hdf5_vars::var_1d_type Type Reference 556
C.62 x_vars::x_1_type Type Reference 558
C.63 x_vars::x_2_type Type Reference 562
C.64 hdf5_vars::xml_str_type Type Reference 567

Appendix D File Documentation **569**
D.1 Documentation/faq.dox File Reference 569
D.2 Documentation/inputs.dox File Reference 569
D.3 Documentation/installation.dox File Reference 569
D.4 Documentation/mainpage.dox File Reference 569
D.5 Documentation/outputs.dox File Reference 570
D.6 Documentation/overview.dox File Reference 570
D.7 Documentation/tutorial.dox File Reference 570
D.8 /Users/toonweyens/Code/PB3D/Modules/driver_eq.f90 File Reference 570

D.9 driver_eq.f90 570
D.10 /Users/toonweyens/Code/PB3D/Modules/driver_POST.f90 File Reference 577
D.11 driver_POST.f90 577
D.12 /Users/toonweyens/Code/PB3D/Modules/driver_sol.f90 File Reference 596
D.13 driver_sol.f90 596
D.14 /Users/toonweyens/Code/PB3D/Modules/driver_X.f90 File Reference 603
D.15 driver_X.f90 603
D.16 /Users/toonweyens/Code/PB3D/Modules/dtorh.f90 File Reference 611
D.17 dtorh.f90 612
D.18 /Users/toonweyens/Code/PB3D/Modules/eq_ops.f90 File Reference 619
D.19 eq_ops.f90 621
D.20 /Users/toonweyens/Code/PB3D/Modules/eq_utilities.f90 File Reference 710
D.21 eq_utilities.f90 711
D.22 /Users/toonweyens/Code/PB3D/Modules/eq_vars.f90 File Reference 723
D.23 eq_vars.f90 724
D.24 /Users/toonweyens/Code/PB3D/Modules/files_ops.f90 File Reference 732
D.25 files_ops.f90 732
D.26 /Users/toonweyens/Code/PB3D/Modules/files_utilities.f90 File Reference 741
D.27 files_utilities.f90 741
D.28 /Users/toonweyens/Code/PB3D/Modules/grid_ops.f90 File Reference 744
D.29 grid_ops.f90 749
D.30 /Users/toonweyens/Code/PB3D/Modules/grid_utilities.f90 File Reference 769
D.31 grid_utilities.f90 770
D.32 /Users/toonweyens/Code/PB3D/Modules/grid_vars.f90 File Reference 806
D.33 grid_vars.f90 807
D.34 /Users/toonweyens/Code/PB3D/Modules/HDF5_ops.f90 File Reference 811
D.35 HDF5_ops.f90 812
D.36 /Users/toonweyens/Code/PB3D/Modules/HDF5_utilities.f90 File Reference 834
D.37 HDF5_utilities.f90 834
D.38 /Users/toonweyens/Code/PB3D/Modules/HDF5_vars.f90 File Reference 839
D.39 HDF5_vars.f90 840
D.40 /Users/toonweyens/Code/PB3D/Modules/HELENA_ops.f90 File Reference 842
D.41 HELENA_ops.f90 842
D.42 /Users/toonweyens/Code/PB3D/Modules/HELENA_vars.f90 File Reference 861
D.43 HELENA_vars.f90 862
D.44 /Users/toonweyens/Code/PB3D/Modules/input_ops.f90 File Reference 863
D.45 input_ops.f90 863
D.46 /Users/toonweyens/Code/PB3D/Modules/input_utilities.f90 File Reference 882
D.47 input_utilities.f90 883
D.48 /Users/toonweyens/Code/PB3D/Modules/messages.f90 File Reference 886
D.49 messages.f90 887

D.50 /Users/toonweyens/Code/PB3D/Modules/MPI_ops.f90 File Reference 894
D.51 MPI_ops.f90 895
D.52 /Users/toonweyens/Code/PB3D/Modules/MPI_utilities.f90 File Reference 901
D.53 MPI_utilities.f90 902
D.54 /Users/toonweyens/Code/PB3D/Modules/MPI_vars.f90 File Reference 916
D.55 MPI_vars.f90 917
D.56 /Users/toonweyens/Code/PB3D/Modules/num_ops.f90 File Reference 919
D.57 num_ops.f90 919
D.58 /Users/toonweyens/Code/PB3D/Modules/num_utilities.f90 File Reference 927
D.59 num_utilities.f90 930
D.60 /Users/toonweyens/Code/PB3D/Modules/num_vars.f90 File Reference 963
D.61 num_vars.f90 969
D.62 /Users/toonweyens/Code/PB3D/Modules/output_ops.f90 File Reference 972
D.63 output_ops.f90 972
D.64 /Users/toonweyens/Code/PB3D/Modules/PB3D_ops.f90 File Reference 995
D.65 PB3D_ops.f90 996
D.66 /Users/toonweyens/Code/PB3D/Modules/PB3D_utilities.f90 File Reference 1013
D.67 PB3D_utilities.f90 1014
D.68 /Users/toonweyens/Code/PB3D/Modules/rich_ops.f90 File Reference 1017
D.69 rich_ops.f90 1017
D.70 /Users/toonweyens/Code/PB3D/Modules/rich_vars.f90 File Reference 1024
D.71 rich_vars.f90 1025
D.72 /Users/toonweyens/Code/PB3D/Modules/SLEPC_ops.f90 File Reference 1026
D.73 SLEPC_ops.f90 1027
D.74 /Users/toonweyens/Code/PB3D/Modules/SLEPC_utilities.f90 File Reference 1051
D.75 SLEPC_utilities.f90 1052
D.76 /Users/toonweyens/Code/PB3D/Modules/sol_ops.f90 File Reference 1055
D.77 sol_ops.f90 1056
D.78 /Users/toonweyens/Code/PB3D/Modules/sol_utilities.f90 File Reference 1076
D.79 sol_utilities.f90 1077
D.80 /Users/toonweyens/Code/PB3D/Modules/sol_vars.f90 File Reference 1088
D.81 sol_vars.f90 1088
D.82 /Users/toonweyens/Code/PB3D/Modules/str_utilities.f90 File Reference 1090
D.83 str_utilities.f90 1091
D.84 /Users/toonweyens/Code/PB3D/Modules/test.f90 File Reference 1093
D.85 test.f90 1094
D.86 /Users/toonweyens/Code/PB3D/Modules/vac_ops.f90 File Reference 1112
D.87 vac_ops.f90 1113
D.88 /Users/toonweyens/Code/PB3D/Modules/vac_utilities.f90 File Reference 1142
D.89 vac_utilities.f90 1143
D.90 /Users/toonweyens/Code/PB3D/Modules/vac_vars.f90 File Reference 1149

D.91 vac_vars.f90 1150
D.92 /Users/toonweyens/Code/PB3D/Modules/VMEC_ops.f90 File Reference 1154
D.93 VMEC_ops.f90 1155
D.94 /Users/toonweyens/Code/PB3D/Modules/VMEC_utilities.f90 File Reference 1159
D.95 VMEC_utilities.f90 1160
D.96 /Users/toonweyens/Code/PB3D/Modules/VMEC_vars.f90 File Reference 1164
D.97 VMEC_vars.f90 1165
D.98 /Users/toonweyens/Code/PB3D/Modules/X_ops.f90 File Reference 1167
D.99 X_ops.f90 1168
D.100 /Users/toonweyens/Code/PB3D/Modules/X_utilities.f90 File Reference 1213
D.101 X_utilities.f90 1214
D.102 /Users/toonweyens/Code/PB3D/Modules/X_vars.f90 File Reference 1218
D.103 X_vars.f90 1219
D.104 PB3D.f90 1226
D.105 POST.f90 1229

Appendix E Examples **1231**
E.1 PB3D.input 1231
E.2 POST.input 1231
E.3 example.mk 1232

Bibliography **1236**

Chapter 1

Welcome to PB3D

The main presentation of the PB3D project information can be found at the `PB3D home`.

Be sure to also consult the `html version` of this document.

This page contains the technical documentation of the **PB3D** project. A pdf version can be downloaded `here`.

It contains a general overview of the code in General code structure, which is much more detailed than what can be found in [17].

To get started, for installation instructions, please see Installation instructions.

Also, in Tutorial, you can find a hands-on tutorial on running the code.

Furthermore, in Input variables, the multitude of input variables is discussed.

Additionally, in Code outputs, the same is done for the output that can be produced, including the output by POST.

For a list of frequently asked questions and frequently encountered problems, consult the Frequently Asked Questions.

Finally, the various modules of which the program consists can be explored in detail in the appendices pages, as well as the procedures and the variables.

—this documentation was created with `Doxygen 1.16.1`—

Chapter 2

Input variables

Footnotes are situated at the end of the chapter

This page describes the various input variables that PB3D and POST can take.

Inputs are generally done through the input file, but there are some inputs that are provided on runtime, using the command-line with `--[option_name]` format.

See also

See init_files().

For both PB3D and POST, both types are discussed.

Some general remarks:

- There is some fault detection built into the routines. These distinguish between minor faults, that trigger a warning but do not stop execution, and major faults, that trigger an error that stop execution.

  See also

  See writo().

- The data type `real` refers to double precision real, defined by num_vars.dp. For integers, standard precision is used.
- There are hard-coded `min_tol` and `max_tol` in read_input_opts(), which are used to limit the tolerances `tol_rich`, `tol_zero` and `tol_SLEPC`. This could be changed if needed.
- If `eq_style = 2` (i.e. HELENA), the total maximum memory available must be enough to perform the first Richardson level in a single equilibrium job (see General code structure).

2.1 PB3D Inputs

2.1.1 Input file

Table 2.1: Table 1. PB3D input file

input parameter	explanation	default value	data type	note
concerning solution				

input parameter	explanation	default value	data type	note
`n_r_sol`	number of points in solution grid	100	`int`	
`min_r_sol`	minimum normalized flux of computational domain $(0 \dots 1)$	0.1	`real`	(1)
`max_r_sol`	maximum normalized flux of computational domain $(0 \dots 1)$	1.0	`real`	(1)
`tol_norm`	tolerance for normal range	0.05	`real`	(2)
`EV_style`	style for EV calculation:↩ 1 SLEPC solver	1	`int`	
concerning field line				
`alpha_style`	style for alpha:↩ 1 one field line, many turns 2 many field lines, one turn	2 (VMEC), 1 (HELENA)	`int`	
`n_alpha`	number of field lines for `alpha_style` 2	10 (VMEC)	`int`	
`alpha`	field line label $[\pi]$ for `alpha_style` 1	0	`real`	
`min_alpha`	minimum field line label α $[\pi]$ for `alpha_style` 2	0	`int`	
`max_alpha`	maximum field line label α $[\pi]$ for `alpha_style` 2	2		

`int`

`concerning perturbation`

`min_n_par_X`

`minimum number of parallel points for integration`

20

`int`

(3)

`min_par_X`

`minimum parallel angle` $[\pi]$ `for integration`

-4

`int`

`max_par_X`

`maximum parallel angle` $[\pi]$ `for integration`

4

int

prim_X

primary mode number in geodesic direction

20

int

(4)

min_sec_X

minimum secondary mode number in parallel direction

int

(4, 5)

max_sec_X

maximum secondary mode number in parallel direction

int

(4, 5)

n_mod_X

number of secondary modes in parallel direction

20

int

(4, 5)

use_pol_flux_F

on which flux to base the normal coordinate.

.true.	rescaled poloidal flux $\frac{\Psi_{\mathtt{pol}}}{2\pi}$
.false.	rescaled toroidal flux $\frac{\Psi_{\mathtt{tor}}}{2\pi}$

.true.

log

(35)

concerning normalization

rho_0

Normalization factor for density ρ $\frac{kg}{m^3}$

10^{-7}

real

R_0

Normalization factor for length scale R m

real

(6)

pres_0

Normalization factor for pressure p Pa

real

(6)

psi_0

Normalization factor for flux Ψ Tm^2

real

(6)

B_0

Normalization factor for magnetic field $\vec{B}$ T

real

(6)

T_0

Normalization factor for time s

real

(6)

concerning input and output

n_sol_requested

number of solutions requested

1

int

retain_all_sol

retain all solutions

.true.	unphysical eigenvalues are also retained

.false.	unphysical eigenvalues are left out

2

log

(7)

plot_resonance

plot the safety factor (poloidal flux) or rotational transform (toroidal flux) and the resonant values

.false.

log

(13)

plot_magn_grid

plot magnetic field lines in the flux surfaces in 3-D geometry as an animation

.false.

log

(14)

plot_B

plot magnetic field $\vec{B} = \nabla\alpha \times \nabla\psi \ [T]$

.false.

log

(15)

plot_J

plot current $\vec{J} = \mu_0^{-1}\nabla \times (\nabla\alpha \times \nabla\psi) \ \left[\frac{A}{m^2}\right]$

.false.

log

(16)

plot_kappa

plot curvature $\vec{\kappa} = \frac{\vec{B}}{B} \cdot \nabla \frac{\vec{B}}{B}$ $\left[\frac{1}{m}\right]$ and its components $\kappa_n = \frac{\nabla\psi}{|\nabla\psi|^2} \cdot \vec{\kappa}$ $\left[\frac{1}{Tm^2}\right]$ and $\kappa_g = \frac{\nabla\psi \times \vec{B}}{B^2} \cdot \vec{\kappa}$ [ ]

.false.

log

(17)

plot_flux_q

plot flux quantities q, ι, p $[Pa]$, Ψ_{pol} $[Tm^2]$ and Ψ_{tor} $[Tm^2]$

.false.

log

(18)

n_theta_plot

number of points in plots in θ direction

int

(19)

n_zeta_plot

number of points in plots in ζ direction

int

(19)

min_theta_plot

minimum of θ range for plots

1

int

max_theta_plot

minimum of θ range for plots

3

int

min_zeta_plot

minimum of ζ range for plots

int

(20)

max_zeta_plot

minimum of ζ range for plots

int

(20)

plot_size

size of 2-D plots in inch by inch

$[10, 5]$

int(2)

ex_plot_style

style how external plots are made

1	GnuPlot

2	Bokeh for 2-D and Mayavi for 3-D

1

int

(21)

concerning Richardson extrapolation

max_it_rich

maximum number of Richardson extrapolations

1

int

tol_rich

tolerance for convergence of Richardson extrapolations

1^{-4}

real

rich_restart_lvl

Richardson level at which to restart a previous simulation

1

int

(8)

concerning finding zeros

max_it_zero

maximum number of iterations in num_ops.calc_zero_hh() and calc_zero_zhang()

100

int

(9)

tol_zero

tolerance for zero finding

1^{-10}

real

(9)

max_nr_backtracks_HH

maximum number of times the correction can be relaxed by half in a backtrack for the Householder methods

20

int

(9)

concerning runtime

use_normalization

use normalization for the internal calculations

.true.

log

(6)

max_tot_mem

maximum total memory available to the simulation $[kB]$

6000

real

sol_n_procs

Number of processes available for SLEPC solution $(1 \dots N)$ where N is the number of MPI processes

N

int

norm_disc_prec_eq

normal discretization precision for equilibrium quantities

3

int

norm_disc_prec_X

normal discretization precision for perturbation quantities

3

int

norm_disc_prec_sol

normal discretization precision for solution quantities

3

int

norm_disc_style_sol

style how solution quantities are discretized

1	central finite differences

2	left finite differences

2

int

magn_int_style

style how magnetic integrals are calculated

1	trapezoidal rule
2	Simpson 3/8 rule

1

int

X_grid_style

style how perturbation grid is set up

1	identical to equilibrium grid
2	identical to solution grid
3	enriched version of equilibrium grid (see max_njq_change)

1 (X_style 1), 2 (X_style 2)

int

(5, 36)

max_njq_change

maximum change of resonant mode numbers for X_grid_style 3

0.49

real

(36)

max_it_SLEPC

maximum number of iterations in SLEPC calculation to find eigenvalue

1000

int

EV_BC

eigenvalue used in boundary condition of style BC_style = 1

1

real

(10)

EV_guess

target guess for eigenvalue

-0.3

real

tol_SLEPC

tolerance used in SLEPC for different Richardson levels

$[1^{-5}, ...]$

real(max_it_rich)

rho_style

style how the density ρ is calculated

1	constant rho_0

1

int

U_style

style how to calculate geodesical perturbation $U = \vec{\xi} \cdot \frac{\nabla\psi \times \vec{B}}{|\nabla\psi|^2}$ is calculated

1	up to order $\sim \frac{1}{n}$

2	up to order $\sim \left(\frac{1}{n}\right)^2$
3	up to order $\sim \left(\frac{1}{n}\right)^3$

3

int

K_style

style how to calculate kinetic energy $K = \int \left|\vec{\xi}\right|^2 dV$

1	normalization of full perpendicular component
2	normalization of only normal component

1

int

BC_style

style how boundary conditions are applied for left and right boundary

1	set to zero
2	minimization of surface energy through asymmetric finite differences
3	minization through extension of grid
4	explicit introduction of the surface energy minimization

$[1, 2]$

int(2)

(10)

norm_style

style how to calculate normalization

1	MISHKA normalization with magnetic field on axis

2	COBRA normalization with pressure on axis

1

int

(11)

solver_SLEPC_style

style used in SLEPC to solve the eigenvalue equation

1	Krylov-Schur with shift-invert
2	generalized Davidson with preconditioner

1

int

matrix_SLEPC_style

style used for SLEPC matrices

1	sparse matrices
2	shell matrices

1

int

(12)

2.1.2 Command-line inputs

Table 2.15: Table 2. PB3D command-line options

input parameter	explanation	argument	note
test	test mode Note Debug version only		(28)

input parameter	explanation	argument	note
no_plots	do not produce any output plots		
no_outputs	do not produce text output		(29)
do_execute_command_line	run the external command line shell commands during execution		(30)
mem_info	produce memory profiles for each process Note Debug version only		(31)
no_guess	do not use a guess		
jump_to_sol	jump straight to the solution for the Richardson level at which the simulationes are (re)started, possibly for different solution grid parameters such as n_r_sol		(33)
export_HEL	create a VMEC input file from HELENA variables, possibly adding a toroidal ripple		(34)
plot_VMEC_modes	plot decay of VMEC modes		
invert_top_bottom_H	invert top and bottom of HELENA equilibrium Note Debug version only		

2.2 POST Inputs

2.2.1 Input file

Table 2.16: Table 3. POST input file

input parameter	explanation	default value	data type	note
concerning solution				
min_r_plot	minimum normalized flux of output domain $(0 \ldots 1)$		real	(24)
max_r_plot	maximum normalized flux of output domain $(0 \ldots 1)$		real	(24)
concerning input and output				
n_sol_plotted	number of solutions plotted	n_sol_requested	int	
pert_mult_factor_POST	factor by which the grid can be perturbed, with respect to the maximum perturbation of each solution independently	0	real	(25)
plot_resonance	plot the safety factor (poloidal flux) or rotational transform (toroidal flux) and the resonant values	.false.	log	(13)

input parameter	explanation	default value	data type	note
plot_magn_grid	plot magnetic field lines in the flux surfaces in 3-D geometry as an animation	.false.	log	(14)
plot_B	plot magnetic field $\vec{B} = \nabla\alpha \times \nabla\psi$ $[T]$	.false.	log	(15)
plot_J	plot current $\vec{J} = \mu_0^{-1}\nabla \times (\nabla\alpha \times \nabla\psi)$ $\left[\frac{A}{m^2}\right]$	.false.	log	(16)
plot_kappa	plot curvature $\vec{\kappa} = \frac{\vec{B}}{B} \cdot \nabla \frac{\vec{B}}{B}$ $\left[\frac{1}{m}\right]$ and its components $\kappa_n = \frac{\nabla\psi}{\lvert\nabla\psi\rvert^2} \cdot \vec{\kappa}$ $\left[\frac{1}{Tm^2}\right]$ and $\kappa_g = \frac{\nabla\psi\times\vec{B}}{B^2} \cdot \vec{\kappa}$ $[\,]$	.false.	log	(17)
plot_flux_q	plot flux quantities q, ι, p $[Pa]$, Ψ_{pol} $[Tm^2]$ and Ψ_{tor} $[Tm^2]$	.false.	log	(18)
plot_sol_xi	plot normal mode $\vec{\xi}$ $[m]$ and its components $X = \frac{\nabla\psi}{B^2} \cdot \vec{\xi}$ $\left[\frac{m^2}{T}\right]$ and $U = \frac{\nabla\psi\times\vec{B}}{\lvert\nabla\psi\rvert^2} \cdot \vec{\xi}$ $[\,]$	.false.	log	(26)
plot_sol_Q	plot magnetic field perturbation due to the normal mode $\vec{Q} = \nabla \times \left(\vec{\xi} \times \vec{B}\right)$ $[T]$ and its components $Q_n = \frac{\nabla\psi}{B^2} \cdot \vec{Q}$ $[m]$ and $Q_g = \frac{\nabla\psi\times\vec{B}}{\lvert\nabla\psi\rvert^2} \cdot \vec{Q}$ $\left[\frac{T}{m}\right]$	.false.	log	(27)
plot_E_rec	plot energy reconstruction	.false.	log	(27)
n_theta_plot	number of points in plots in θ direction		int	(19)
n_zeta_plot	number of points in plots in ζ direction		int	(19)
min_theta_plot	minimum of θ range for plots			

<_form>

int

max_theta_plot

minimum of θ range for plots

3

int

min_zeta_plot

minimum of ζ range for plots

int

(20)

max_zeta_plot

minimum of ζ range for plots

int

(20)

plot_size

size of 2-D plots in inch by inch

$[10, 5]$

int(2)

ex_plot_style

style how external plots are made

1	GnuPlot
2	Bokeh for 2-D and Mayavi for 3-D

1

int

(21)

concerning Richardson extrapolation

PB3D_rich_lvl

Richardson level at which to perform post-processing

int

(22)

concerning finding zeros

max_it_zero

maximum number of iterations in num_ops.calc_zero_hh() and calc_zero_zhang()

100

int

(9)

tol_zero

tolerance for zero finding

1^{-10}

real

(9)

max_nr_backtracks_HH

maximum number of times the correction can be relaxed by half in a backtrack for the Householder methods

20

int

(9)

concerning runtime

max_tot_mem

maximum total memory available to the simulation $[kB]$

6000

real

POST_style

style how post-processing is done

1	extended rectangular grid in θ and ζ
2	field-aligned grid used internally by PB3D

1

int

(23)

plot_grid_style

style used for output plotting

0	3-D plots

1	slab plots: θ and ζ are sides of a slab
2	slab plots with folding of fundamental interval $0 \ldots 2\pi$
3	straight cylinder geometry: ζ is straightened

0

int

(23)

2.2.2 Command-line inputs

Table 2.20: Table 4. POST command-line options

input parameter	explanation	argument	note
test	test mode Note Debug version only		(28)
no_plots	do not produce any output plots		
no_outputs	do not produce text output		(29)
do_execute_command_line	run the external command line shell commands during execution		(30)
mem_info	produce memory profiles for each process Note Debug version only		(31)
swap_angles	swap θ and ζ of output grid in POST_style = 2		
compare_tor_pos	compare $\vec{B}$, $\vec{J}$ and $\vec{\kappa}$ from plot_B, plot_J and plot_kappa at different toroidal positions, as well as the difference in position		

RZ_0(2)

(32)

Note

1. The normalized flux is either the poloidal (use_pol_flux_F = .true.) or toroidal (use_pol_flux_F = .false.), divided by its value at the boundary.
2. Is used in check_x_modes() whether there exists a range in which each of the modes resonates.
3. This is the number for the first Richardson level. It is doubled for every next level.

4. Should satisfy the high- n approximation, currently set to > 5 .
5. If min_sec_X and max_sec_X is set, the user prescribes the secondary mode number range (X_style = 1). If n_mod_X is set, on the other hand, the user prescribes the size of the range, which is automatically chosen (X_style = 2). They cannot be both chosen.
6. Normalization constants, with exception of rho_0 are set in the routine calc_normalization_const().
7. In store_results(), it is decided whether the eigenvalue is physical or not, by considering the ratio between real and imaginary part.
8. This needs to be between 1 and the maximum level that was obtained in a previous solution, plus 1. For technical HDF5 reasons, if Richardson restart is used, the same input fil parameters should be used for max_tot_mem and the same number of processes should be used. Furthermore, it should not exceed max_it_rich.
9. For the case of num_ops.calc_zero_hh(), at every iteration it is checked whether the correction approximation is better than the original one. If it is not, the correction is relaxed by a factor half. This is done max_nr_backtracks_HH times.
10. See set_bc().
11. See calc_normalization_const().
12. See solve_ev_system_slepc()
13. See resonance_plot().
14. See magn_grid_plot().
15. See b_plot(). The output is done in various coordinate system, as explained in calc_vec_comp().
16. See j_plot(). The output is done in various coordinate system, as explained in calc_vec_comp().
17. See kappa_plot(). The output is done in various coordinate system, as explained in calc_vec_comp().
18. See flux_q_plot().
19. For eq_style = 1, the equilibrium is axisymmetric, so by default n_theta_plot = 501 and n_zeta_plot = 1; for eq_style = 2, the equilibrium can be 3-D, so n_theta_plot = 201 and n_zeta_plot = 51
20. For eq_style = 1, the equilibrium is axisymmetric, so by default min_zeta_plot = max_zeta_plot 0; for eq_style = 2, the equilibrium can be 3-D, so min_zeta_plot = 0 and max_zeta_plot = 2.
21. This refers to the external plots, which are complimentary to the HDF5 plots. See draw_ex().
22. If output is done that requires a solution, this needs to be between 1 and the maximum level that was obtained in PB3D. See find_max_rich_lvl(). If no solution is found, some of the outputs that do not depend on this still work.Optionally, this can be forced by choosing PB3D_rich_lvl not positive.
23. The POST_style determines whether the post-processing is done on an extended, new grid, or on the PB3D internal grid. On top of this, through plot_grid_style the user can set the output grid for the post-processing. Note that some combinations of styles do not make sense.
24. By default, the plot normal range is chosen equal to the solution normal range from min_r_sol and max_r_sol, but it can be restricted. This is useful, for example, to reduce computation time.
25. The Cartesian components of the result of the division of pert_mult_factor_POST by the maximum value of the solution normal mode amplitude is added to the equilibrium Cartesian grid components to illustrate the perturbation in 3-D. This is done for each solution independently and sequentially. If it is not positive, it is ignored. Note that plot_sol_xi = .true. is necessary for this to be used.

26. These quantities are calculated in sol_utilities.calc_xuq(). This is done for a time sequene, currently hard-coded in plot_sol_vec(), depending on whether the solution is stable or unstable.
27. The energy reconstruction is performed in calc_e() if either one of plot_sol_xi, plot_sol_Q or plot_E_rec is true.
28. Equivalent use is through -t. If this is used, do_execute_command_line should be set to true as well. See generic_tests().
29. This is used internally as well, to suppress certain output. See run_driver_x() and run_driver_post().
30. This can fail on local systems, and generally works badly on computational clusters, where the computational nodes often don't have the external tools (see ex_plot_style) installed. By default, an output file is created for the shell commands, which can be executed afterwards. See Shell commands file.
31. This is done by calling get_mem_usage() and storing the result at every allocation and deallocation of an eq_1_type, eq_2_type, X_1_type, X_2_type, vac_type or sol_type, as well as for input quantities. Furthermore at every user output with writo(), this is also done, and the output is prepended to the output string. Also, the output is written to a file. See Memory usage file.
32. This is useful to calculate the ripple in these quantities. To use this, it is necessary to have an output grid of only 3 points, where the poloidal projection of the middle point should be the in the middle between the poloidal projections of the two others. Also, you have to provide through RZ_0 the R and Z value of the origin of the geometrical poloidal angle that is used to calculate distances. See grid_utilities.calc_tor_diff().
33. Jumping straight to the solution is useful if a simulation already calculated all the necessary quantities to set up the solution matrices $\overline{\mathsf{A}}$ and $\overline{\mathsf{B}}$ in $\overline{\mathsf{A}}\vec{X} = \lambda\overline{\mathsf{B}}\vec{X}$ and the solution needs to be redone, for example because a calculation was attempted with a method that did not converge.
34. This is a rather complicated procedure with many possibilities that is intended for expert use. See create_vmec_input().
35. Using the toroidal flux is **experimental** and **untested**.
36. Have a look in calc_norm_range(). X_grid_style 1 is generally too coarse while X_grid_style 2 is too fine. X_grid_style 3 was designed to have the both of best world. It only makes sense for X_style 2 (fast), though, as in the enrichment process the fast secondary mode range is used to limit the maximum amount the quantity nq (when using poloidal flux) or $m\iota$ (when using toroidal flux) to max_njq_change between two normal grid points.

Chapter 3

Frequently Asked Questions

3.1 runtime errors

Table 3.1: Table 1. frequently encountered errors

error	solution
”need at least N points in grid”	grid_utilities.setup_deriv_data() needs a minimum number of points in a grid to setup data with which the derivatives can be calculated later in grid↩ _utilities.apply_disc(). If the grid does not have enough points, and the order of the derivative is too high, the procedure gives this error message. The solution is to either increase the number of points, or to lower the order. It might be that the number of points in the grid is lower than you expect because the grid is divided. In this case, you need more memory (see `max_tot_mem` in Input variables).

error	solution
”Aument number of modes or choose finer grid”	The range of secondary mode numbers at every flux surface is either user-specified, for `X_style` 1, or chosen automatically around the most resonant values for `X_style` 2 (see Input variables). This implies that, on the other hand, every secondary mode should have a normal range in which it is present, possibly multiple ranges for reversed-shear configurations. The code automatically checks whether this is indeed the case. It could, for example, be that a certain mode number is not present anywhere, when the normal grid in which it is tabulated is too coarse. A solution is therefore to choose a finer grid. Alternatively, for `X_style` 1, the range can be adapted, and for `X_style` 2 the number of modes `n_mod_X`. Note that the particular grid that this discussion is about, can be found out by the trace of the error message. If it results from the solution driver, it concerns the solution grid, and if it concerns the perturbation driver, it concerns the perturbation grid. The solution grid is specified by its range through `min_r_sol` and `max_r_sol` and the number of points `n_r_sol`, and the perturbation grid by the `X_grid_style`. See Input variables.
”Zero pivot in LU factorization”	There exist cases in which the eigenvalue solver fails when `n_mod_X = 1`, for unkown reasons. Increase `n_mod_X` or play with other parameters, such as the `norm_disc_prec_sol`.
`jump_to_sol` does not work	Jumping over the equilibrium and perturbation drivers straight to the solution only works if solution parameters such as the boundary conditions are changed, *not* if the number of points in the solution, `n_r_sol`, is changed, as in this case the equilibrium and solution drivers need to be recalculated properly.
The solution suffers from numerical oscillations	This is a work in progress. Change the limits of the solution grid `min_r_sol` (and possibly `max_r_sol`). Better even is to use asymmetric finite differences `norm_disc_style_sol = 2`

3.2 common problems

Table 3.2: Table 2. common problems

problem	solution
code compilation stuck	Especially on intel systems, it has been found that sometimes the compiler seems to be stuck, especially for `eq_ops`. A solution is to use less optimization for the problematic files through `-O2`.

problem	solution
unphysical eigenvector	For coarse equilbrium grids and/or strongly varying safety factors, and when using the fast version (i.e. with `n_mod_X`), it is possible that the resonating mode number range varies too quickly, leading to inaccurate interpolation of tensorial perturbation quantities in the solution grid. Try using `X_grid_style` 3, where enrichment of the equilibrium grid is used to set up the perturbation grid. If this does not help, additionally lower `max_njq_change` from its default value.
low accuracy for energy reconstruction	Energy reconstruction tries to see whether by calculating the total volume integral of the perturbed potential energy and dividing it by the total volume integral of the kinetic energy of the perturbation, the eigenvalue can be recovered. This gives a lot of insight in the processes that drive the instabilities, by looking at the individual terms of these integrals, and is a good check on the internal consistency of the PB3D code. Note that for axisymmetric equilibria, the volume integral reduces to a surface integral in a poloidal cross-section. It is actually very hard to perform a good energy reconstruction because the techniques used differ from the ones in the PB3D code, where the normal derivatives of the eigenvector are performed using finite differences, while in the energy reconstruction direct volume integrals are used. To get the best possible results, therefore, it is important to achieve full Richardson convergence in the number of parallel points (by increasing `max_it_rich` and/or `min_n_par_X`), as well as convergence in the number of secondary mode harmonics (by increasing `n_mod_X`). Doing this, for axisymmetric cases, the code should retun quite accurate results, from version 2.26 upwards. These can furthermore be improved by choosing a finer post-processing grid (see `n_theta_plot` and `n_zeta_plot`). It is, however, possible that a small fraction, possibly of the order of a percent remains in the relative difference. This is probably due to the fact that in PB3D for `X_style` 2 (fast), the stencil is cut when the normal index falls out of its mode range (see insert_block_mat()).
Changing input does not work for Richardson extrapolation	You cannot change the inputs as Richardson extrapolation depends on having these constant (with the exception of the number of poins in the parallel integrals, but this is done internally in PB3D). The important exception to this fact, however, is that you can change some of the solution grid parameters, such as `n_r_sol` (see Input variables).

problem	solution
The code has not been tested satisfactorily for decreasing normal coordinate.	When the normal coordinate is decreasing (but still monotonous), the code should be able to handle this. However, a quick test gave some problems, which have to be debuged. An easy workaround is to just run the equilibrium solver again with inverted fields and currents.
The code has to be adapted for reversed-shear regions still.	When there are reversed-shear regions, the routines that calculate the resonant flux surfaces, such as `calc_res_surf()` need to be extended. If this is not done, there will be problems, mostly for `X_style` 2 (fast), and especially for `X_grid_style` 3 (enriched).

Chapter 4

Code outputs

Footnotes are situated at the end of the chapter

This page describes the various outputs that PB3D and POST can produce.

On the one hand, there are various text output files, which are discussed in section Output Files.

Furthermore, a multitude of plots can be produced. This is the topic of section Output Plots.

4.1 Output Files

Some of these files are opened in open_output(), but others are opened afterwards.

4.1.1 Log file

Both PB3D and POST produce a log file, in addition to what is produced on the screen.

These log files are called `PB3D_out.txt` and `POST_out.txt` and they contain the same information as the output on the screen, but without the special formatting provided by the foul library (see Introduction).

4.1.2 Shell commands file

Some output routines make use of use_execute_command_line() in order to execute shell commands. This can be, for example, the creation of a plot by calling GnuPlot, among other things.

However, this often does not work well by default (see 30 in Input variables). In PB3D and POST, therefore, by default, these commands are not executed, but written to an output file. After execution of PB3D and POST has finished, this file can be made executed.

This behavior can be changed using the `--do_execute_command_line` flag (see Input variables).

4.1.3 HDF5 output file

PB3D writes variables to an HDF5 output file during execution, as shown in table 4.1. Note that there is a difference between the equilibrium styles because of the way how the equilibrium codes provide output:

- VMEC (`eq_style = 1`) produces double Fourier series:
  - This can be evaluated at arbitrary angular points.
  - In PB3D, the equilibrium and perturbation variables are therefore already calculated on the field-aligned grid.
  - Parallel integration can then be done directly.
- HELENA (`eq_style = 2`) produces 2-D tables for the poloidal cross-section.
  - This needs to be interpolated along the magnetic field lines before parallel integration.
  - The equilibrium and perturbation variables are calculated first in the HELENA grid.

Table 4.1: Table 1. HDF5 output file

variable type	VMEC (`eq_style = 1`)	HELENA (`eq_style = 2`)
inputs	in preliminary phase	
grids of grid_vars.grid_type	**equilibrium**	
	field-aligned-grid at every Richardson level	
		also output HELENA grid at first Richardson level
		used later for interpolation to field-aligned grid
	perturbation	
	field-aligned-grid at every Richardson level	
		also output HELENA grid at first Richardson level
		used later for interpolation to field-aligned grid
	solution	
	only at first Richardson level	
flux equilibria of eq_vars.eq_1_type	only at first Richardson level	
metric equilibria of eq_vars.eq_2_type	only at first Richardson level	
		also output HELENA grid at first Richardson level
		used later for interpolation to field-aligned grid
vacuums of vac_vars.vac_type	at every Richardson level	only at first Richardson level
solutions of sol_vars.sol_type	at every Richardson level	

This data is used for Richardson restart (see variable `rich_restart_lvl` in Input variables).

In the post-processing phase, POST reads this data as well.

4.1.4 Eigenvalue summary file

In each Richardson extrapolation level, after the solution of the eigenvalue system of equations, eigenvalues are stored in PB3D.

- For SLEPC, the results are saved in the procedure store_results():
  - If normalization is used, the normalization time scale is also stored.
  - If an unphysical complex eigenvalue is detected, this is indicated. See `option` retain_all_sol in Input file.

4.1.5 Energy reconstruction file

If energy reconstruction is performed by POST (see `plot_E_rec` in Input variables), the individual terms that make up the perturbed potential energy as well as the kinetic energy of the perturbation are calculated and plot. (3) Furthermore, the integration of these quantities over the whole volume, is returned in an output file.

In table 4.2, an overview is given of the different terms. Here, ρ is the density, $\sigma = \frac{\vec{B}\cdot\vec{J}}{B^2}$ is the parallel current and $\kappa = \frac{\vec{B}}{B}\cdot\nabla\frac{\vec{B}}{B}$ is the magnetic curvature. See calc_e().

Table 4.2: Table 2. energy reconstruction

potential energy E_{pot}	kinetic energy E_{kin}
• normal line bending term $\frac{1}{\lvert\nabla\psi\rvert^2}\left(\frac{\nabla\psi}{\lvert\nabla\psi\rvert^2}\cdot\vec{Q}\right)^2$ • gedesic line bending term $\mathcal{J}^2\frac{\lvert\nabla\psi\rvert^2}{B^2}\left(\frac{\nabla\psi\times\vec{B}}{B^2}\cdot\vec{Q}\right)^2$ • normal ballooning term $-2p'\left(\nabla\psi\cdot\vec{\xi}\right)^2\left(\nabla\psi\cdot\vec{\kappa}\right)$ • geodesic ballooning term $-2p'\left(\nabla\psi\cdot\vec{\xi}\right)\left(\frac{\nabla\psi\times\vec{B}}{\lvert\nabla\psi\rvert^2}\cdot\vec{\xi}\right)^*\left(\frac{\nabla\psi\times\vec{B}}{\lvert\nabla\psi\rvert^2}\cdot\vec{\kappa}\right)$ • normal kink term $-2\sigma\left(\nabla\psi\cdot\vec{\xi}\right)^*\left(\frac{\nabla\psi\times\vec{B}}{B^2}\cdot\vec{Q}\right)$ • geodesic kink term $2\sigma\left(\frac{\nabla\psi\times\vec{B}}{B^2}\cdot\vec{\xi}\right)^*\left(\nabla\psi\cdot\vec{Q}\right)$	• normal kinetic term $\frac{1}{\lvert\nabla\psi\rvert^2}\left(\frac{\nabla\psi}{\lvert\nabla\psi\rvert^2}\cdot\vec{\xi}\right)^2$ • gedesic kinetic term $\mathcal{J}^2\frac{\lvert\nabla\psi\rvert^2}{B^2}\left(\frac{\nabla\psi\times\vec{B}}{B^2}\cdot\vec{\xi}\right)^2$

4.1.6 Memory usage file

The memory usage of PB3D and POST can be monitored using the optional input argument `--mem_info` (see Command-line inputs).

This is a rather crude implementation and more advance profiling methods should be used, such as `Scalasca` on top of `Score-P`.

Note

Debug version only

4.2 Output Plots

A multitude of plots can be produced by PB3D and POST.

POST can produce all the plots that PB3D can, and some more.

The plots themselves fall in two categories:

- Plots using an external program.
  - Produced by output_ops.print_ex_2d() or output_ops.print_ex_3d() combined with draw_ex().
  - A list of external programs is given in draw_ex().
  - These plots require a shell command to be run, which is disabled by default and the commands are stored in a Shell commands file.
  - The produced plots can be 2-D or 3-D, and they can be interactive or not, depending on the `ex_plot_style` (see Input variables).
- Data files that can be used to produce plots with visualization software:
  - The software used here was `VisIt`, based on personal preference, but `ParaView` or other VTK-based solutions should work equally well.
  - Every plot creates two output files: one heavy-data HDF5 file and one `XDMF` file that contains specifics about the HDF5 file.
  - This is necessary, as HDF5 itself is too general a format for general visualization tools to interpret easily.
  - The XDMF file that accompanies each HDF5 file basically contains information about the saved variables, such as whether they are vectors or scalars, and on which grid type they are defined.
  - This is the file that should be opened by the visualzation software.
  - The dimensions of these plots can be 2-D or 3-D, depending on `eq_style`. See 19 and 20 in Input variables.

An overview of possible plots is now given in table 4.3. The name of the respective input parameter is provided as well. See Input variables.

Table 4.3: Table 3. possible plots

plot type	external plot	HDF5 plot
flux quantities (`plot_flux_q`)	• pressure p • poloidal flux Ψ_{pol} • toroidal flux Ψ_{tor}	• safety factor q • rotational transform ι • pressure p • poloidal flux Ψ_{pol} • toroidal flux Ψ_{tor}
resonance (`plot_resonance`)	safety factor q (poloidal flux) or rotational transform ι (toroidal flux) with the indication of the resonant flux surfaces (1)	surfaces of resonant values for safety factor q (poloidal flux) or rotational transform ι (toroidal flux) (1)

plot type	external plot	HDF5 plot
magnetic grid (`plot_magn_grid`)		magnetic grid lines in the magnetic flux surfaces, as a time collection
magnetic field $\vec{B}$ (`plot_B`)		co- and contravariant components of magnetic field, as well as magnitude and vector plot
current density $\vec{J}$ (`plot_J`)		co- and contravariant components of current density, as well as magnitude and vector plot
magnetic curvature $\vec{\kappa}$ (`plot_kappa`)		co- and contravariant components of magnetic curvature, as well as magnitude and vector plot
solution normal mode $\vec{\xi}$ (`plot_sol_xi`)		real and imaginary part and phase of perturbation of position, as well as vector plot
perturbated magnetic field due to solution normal mode $\vec{Q}$ (`plot_sol_Q`)		real and imaginary part and phase of perturbation of magnetic field, as well as vector plot
energy reconstruction (`plot_E_rec`)		different components of energy reconstruction, integrated as well as profiles (2)

Note

1. See `use_pol_flux_F` in Input variables.
2. The integrated energy reconstruction terms are also output to a file. See Energy reconstruction file.
3. Energy reconstruction only works well when `POST_style = 2` is used, i.e. when the same grid is used as for PB3D.

Chapter 5

General code structure

This page describes the general code structure of PB3D, as well as its accompanying post-process program POST. It is more complete than the documentation that can be found in [17].

5.1 PB3D flowchart

In figure 1, the top-level flowchart of PB3D is shown (adapted from [17]):

Figure 5.1 Figure 1 : basic PB3D flowchart

Roughly speaking, the main body of the code is taken up by four driver routines:

- input driver
- equilibrium driver
- perturbation driver
- solution driver

The **input driver** sets the other drivers up by reading input options as well as quantities from equilibrium codes, and converting them to the internal PB3D format.

The **equilibrium driver** uses these input variables to set up equilibrium quantities.

- On the one hand these consist of flux equilibrium quantities (stored in a eq_vars.eq_1_type) that only depend on the flux coordinates, such as the pressure p, the safety factor q, etc.
- On the other hand, they consist of metric equilibrium quantities (stored in a eq_vars.eq_2_type) that depend on the angular position as well, such as the metric quantities of the grid $g_{ij} = \vec{e}_i \cdot \vec{e}_j$ and $h_{ij} = \nabla u^i \cdot \nabla u^j$, as well as derived quantities such as the parallel current $\sigma = \frac{\vec{B}\cdot\vec{J}}{B^2}$, etc. [16]

The **perturbation driver** uses both types of equation variables to calculate perturbation quantites. Again, there are two types.

- The first type of variables are the vectorial perturbation quantities (stored in a x_vars.x_1_type), which contains the geodesic component of the perturbation $U = \frac{\nabla\times\vec{B}}{|\nabla\psi|^2}\cdot\vec{\xi}$ where $\vec{\xi}$ is the perturbation. Also the parallel derivatives of these quantities are calculated.
- The other type re the tensorial perturbation quantities (stored in a x_vars.x_2_type), and it combines the vectorial perturbation quantities with the equilibrium quantities to form the tensorial quantities $\widetilde{PV}^i_{k,m}$ and $\widetilde{KV}^i_{k,m}$ that represent the perturbed potential energy as well as the kinetic energy of the perturbation [16].

These are also complemented by $\delta^{\text{vac}}_{k,m}$, the vacuum contribution to the perturbed potential energy (1).

After integrating them along the magnetic field lines, these magnetic integrals form the building blocks of the discretized matrices $\overline{\text{A}}$ and $\overline{\text{B}}$ in the generalized system of eigenvalue equations $\overline{\text{A}}\vec{X} = \lambda\overline{\text{B}}\vec{X}$.

These matrices are set up in the **solution driver**, which currently works with the SLEPc library that is build on the PETSc library for sparse linear algebra. After solution, the eigenvalues and -vectors are stored appropriately.

Finally, it is checked whether another Richardson level should be attempted or not in **Richardson converged?**, and if so, the number of parallel grid points is doubled.

5.2 Equilibrium & Perturbation Jobs

The general code structure explained in PB3D flowchart basically comes down to setting up the matrices to be solved for the eigenvalues and eigenvectors. In general, the dimensions of these variables are too large (see the grids in Detailed PB3D Flowchart) for them to fit in memory. A structure of two nested loops is therefore employed in PB3D, called **jobs**, to keep the memory requirements under a user-defined maximum (see `max_tot_mem` in Input variables).

5.2.1 Perturbation jobs

perturbation jobs are used to divided the calculation of the tensorial perturbation quantities, which come in pairs of mode numbers. These combinations are blocked, where each block fits in memory. These blocks are the perturbation jobs.

They form the inner loop.

See also

See divide_x_jobs().

5.2.2 Equilibrium jobs

As a result the memory still needs to hold all the relevant equilibrium and vectorial perturbation quantities necessary for the individual perturbation jobs during the whole time these jobs are being calculated. **equilibrium jobs** are therefore employed in an outer loop to divide these equilibrium and vectorial perturbation quantities in manageable chunks.

To be more precise, the parallel range over which variables are to be integrated in the magnetic integrals (see Detailed PB3D Flowchart), is cut in pieces. The advantage here is that only the results from the sub-integrals have to be saved and updated, not the function values themselves.

See also

See divide_eq_jobs().

5.3 Detailed PB3D Flowchart

The different blocks if figure 1 are expanded in table 5.1. The cells that are typeset in light blue are only performed for the first equilibrium job of either the restart Richardson level, or the first Richardson level if this is not provided.

Table 5.1: Table 1. detailed PB3D flowchart

variable type	VMEC (`eq_style = 1`)	HELENA (`eq_style = 2`)
PB3D initialize	the PB3D structure is set up, such as the MPI communicator, etc.	
input driver	The arguments are parsed (see parse_args())	
	The input files are opened (see open_input() and Input variables)	
	The user inputs are processed (see read_input_opts())	
	The equilibrium code output is read and converted to the internal PB3D format (see read_inpu	
	Normalization constants are calculated (see calc_normalization_const()) and the input is norm	
	Output files are opened (see open_output() and Code outputs)	
	Input variables are written to HDF5 output (see print_output_in())	
	Some other variables are broadcasted directly to all the processes using MPI (see broadcast_i	
	normal part of equilibrium grid is set up, as for the angular part, the field-lines are necessary which are not yet calculated	the equilibrium grid is set up equal to the HELENA out
	flux equilibrium quantities (see eq_vars.eq_1_type) are calculated (see eq_ops.calc_eq())	
	flux equilibrium quantities are written to HDF5 output (see eq_ops.print_output_eq())	
	equilibrium grid is completed by tracing the field lines (see calc_ang↩ _grid_eq_b())	an additional, field-aligned equilibrium grid is set up l setup_grid_eq_b()), and it is written to HDF5 output (se
	the equilibrium grid is written to output	the equilibrium grid is written to the output

equilibrium driver

variable type	VMEC (`eq_style = 1`)	HELENA (`eq_style = 2`)
	the metric equilibrium variables (see eq_vars.eq_2_type) are calculated (see eq_ops.calc_eq())	the metric equilibrium variables (see eq_vars.eq_2 eq_ops.calc_eq())
		the metric equilibrium variables are writte eq_ops.print_output_eq())
	store equilibrium quantities necessary for vacuum calculation (see store_vac())	store equilibrium quantities necessary for vacuum calc
	possible plot of magnetic field, current density and/or curvature in (field-aligned) equilibrium grid (see b_plot(), j_plot(), kappa_plot())	possible plot of magnetic field, current density and/or librium grid (see b_plot(), j_plot(), kappa_plot())
	redistribute the equilibrium grid (see redistribute_output_grid())	redistribute the equilibrium grid (see redistribute_outp
	redistribute the flux equilibrium quantities (see eq_ops.redistribute_output_eq())	
	redistribute the metric equilibrium quantities (see eq_ops.redistribute_output_eq())	redistribute the metric equilibrium quantities (see eq_
		The field-aligned equilibrium grid is redistributed (see
	possible plot of magnetic field lines in flux surfaces (see magn_grid_plot()) if last equilibrium jo	
		the metric equilibrium quantities are inerpolated in the (see interp_hel_on_grid())
	determine perturbation grid (see setup_grid_x()) and write it to HDF5 ouput (see print_output_grid())	determine perturbation grid (see setup_grid_x()) and print_output_grid())
		set up field-aligned perturbation grid and write to outp
	set up mode number information (see setup_nm_x()) and check them (see check_x_modes()) if f	
	possibly plot resonance (see resonance_plot()) if first perturbation job	
perturbation driver	the vectorial perturbation variables (see x_vars.x_1_type) are calculated (see x_ops.calc_x())	the vectorial perturbation variables (see x_vars.x_ x_ops.calc_x())
		write vectorial perturbation variables to output (see x_
		interpolate vectorial perturbation variables on field-ali _grid())
	calculate tensorial perturbation variables (see x_ops.calc_x())	
	calculate magnetic integrals (see calc_magn_ints())	
	write magnetic integrals to output (see x_ops.print_output_x())	
	calculate vacuum (see calc_vac())	calculate vacuum (see calc_vac())
solution driver	set up the solution grid (see setup_grid_sol()) and write to HDF5 output (see print_output_grid	

variable type	VMEC (`eq_style = 1`)	HELENA (`eq_style = 2`)
	solve system of equations (see solve_ev_system_slepc())	
Richardson converged?	check for convergence of Richardson levels by comparing solution eigenvalues (see check_con	
PB3D finalize	clean up	

5.4 POST Overview

The POST program for post-processing of PB3D results consists of just one driver.

- In the first part, everything is set up (see init_post()):
  - This includes initializing the variables written to HDF5 in PB3D.
  - Furthermore, the output grids are created
  - Also, 1-D flux plots are produced, which are part of the plots that PB3D is able to produce (see `plot_flux_q`, `plot_resonance` and `plot_magn_grid` in table 4.3 in Code outputs) can be created here.
- After that, new outputs are calculated and plot (see run_driver_post()):
  - Quantities are calculated on the output grids.
  - Using these, plots are first produced that do not depend on the solution vector from PB3D, and are also part of the plots that PB3D is able to produce (see `plot_B`, `plot_J`, `plot_kappa` in table 4.3 in Code outputs).
  - Finally, other plots are produced that do depend on these (see `plot_sol_xi`, `plot_sol_Q`, `plot_E_rec` in table 4.3 in Code outputs).

POST also knows the concept of equilibrium jobs, but not that of perturbation jobs (see Equilibrium & Perturbation Jobs). This is necessary for larger outputs that quickly overflow memory. The HDF5 output files are updated at every equilibrium job, but, with reasonable safety, they can already be looked at in a visualizer.

Note

1. This is not yet implemented.

Chapter 6

Tutorial

Footnotes are situated at the end of the chapter

This page contains a hands-on tutorial to get you used to running PB3D to calculate 3-D ideal linear high-n MHD stability, and POST to post-process the results.

Be sure to check out the Installation instruction first.

If at any point a routine is not clear, don't hesitate to consult the appendices.

6.1 Getting the Equilibrium

PB3D calculates stability of an equilibrium magnetic toroidal configuration. It is written in a modular way to isolate this main task from the actual code that was used to obtain the equilibrium configuration, and the format that this code may use. Currently, two formats are possible:

- 3-D VMEC [8].
- Axisymmetric HELENA [11].

VMEC is a general 3-D code built on an interesting idea: The poloidal coordinate θ is deformed in such as a way to make the Fourier basis that is used to describe the cylindrical coordinates $R(s,\theta,\varphi)$ and $Z(s,\theta,\varphi)$ as narrow as possible, where s is the flux label ($s = \sqrt{\Psi_{\text{tor}}/\Psi_{\text{tor,edge}}}$) and φ is the regular cylindrical angle (1). This is done by introducing a deformation factor $\lambda(s,\theta,\varphi)$ defined so that the straight-field line coordinate θ_{F} is given by $\theta_{\text{F}} = \theta + \lambda$.

The fact that R, Z and λ are given as a double Fourier series in θ and ζ ensures that the angular derivatives can be done analytically. This property is used in PB3D. Apart from that, the flux functions that describe the profiles on the flux surfaces, the rotational transform $\iota(s)$ and pressure $p(s)$, are also returned.

For HELENA, the situation is different, as this code considers axisymmetric equilibria only. Its output consists, aside from flux functions safety factor $q(\psi)$, presure $p(\psi)$ and toroidal covariant magnetic field $F(\psi)$, where $\psi = \Psi_{\text{pol}}/\Psi_{\text{pol,edge}}$ is the normalized poloidal angle, of a 2-D table for the upper metric factors $h^{\psi\psi}$ and $h^{\psi\theta}$ and the lower metric factor $g_{\zeta\zeta}$. HELENA uses the same angular coordinates θ and $\zeta = -\varphi$ as PB3D (see 1).

Note

The modular nature of PB3D enables easy integration of other equilibrium codes, if necessary.

To get the appropriate equilibrium code results into PB3D, it suffices to place the output files in the same directory in which PB3D is run:

- for HELENA, it is a text file, corresponding to the mapping file (`fort.12`),
- for VMEC, it is the so-called `wout` file,

and to specify their names as the second input argument, the first one being the file with user inputs (see Setting Up the Input File).

Note

For VMEC, the `wout` file has to be in Netcdf's `.nc` format. The old `.txt` format is no longer supported.

In any case, if you forget how to run the PB3D (or POST) programs, you can always just type

```
./PB3D --help
```

or even just

```
./PB3D
```

Note

The HELENA version used here outputs the variables `IAS` and `B0` to the mapping file, after the variable `JS0` and after the variable `RAXIS`. Some versions do not write these by default. If yours is one of the above, the solution is to change

```
write(nmap,8) JS0
```

to

```
write(nmap,8) JS0, IAS
```

and

```
write(nmap,8) RAXIS
```

to

```
write(nmap,8) RAXIS, B0
```

in the subroutine `mapping` and recompile and re-run HELENA.

6.2 Checking the equilibrium (optional)

It is always important that the equilibrium has been properly calculated, well-converged, etc.

It is possible to use PB3D to check this. This will be quickly described in this optional section.

The checks described below are only possible with the code compiled with the debug flag on (see Installation). Using the debug version, the tests can be triggered using the `--test` option when running PB3D (or POST), and using the interactive user interface.

They generally output HDF5 files that can be read with ParaView or VisIt to check the difference between two plots. After having a look in the code (search for `ltest`), this should be self-explanatory.

Note

Most of the checks are only useful for developers. However, the tests on `B_F` and on the pressure balance consistency are vital.

Table 6.1: Table 1. the most important tests

tests	purpose
calculation of `g_V`	Check whether equilibrium metric coefficients calculated for VMEC equilibria through application of transformation matrices match with a direct implementation of the VMEC metric coefficients. Note that the reason why the transformation matrices are used, is that derivatives can be easily calculated as well, with high precision.

tests	purpose
calculation of `jac_V`	Similar to the test on the calculation of `g_V`, but this time for the Jacobian of VMEC coordinates.
consistency of HELENA metric factors	Check whether the metric factors returned by HELENA are consistent with the ones calculated from the flux quantities, R and Z. Basically, this is a measure for the accuracy with which the pressure balance is satisfied.
harmonic content of HELENA output	Analyze harmonic content of HELENA variables.
calculation of `D1 D2 h_H`	Check whether the cross-derivatives are correctly and accurately calculated.
calculation of `T_EF`	Check is $\mathcal{T}_{\mathsf{E}}^{\mathsf{F}}$ complies with the (unpublished) theory.
calculation of `jac_F`	Similar to the test on the calculation of `jac_V`, but this time for the Jacobian of Flux coordinates.
calculation of `B_F`	Similar to the test on the calculation of `jac_F`, but this time for the covariant components and magnitude of the magnetic field in Flux coordinates.
consistency of pressure balance	Check how well the pressure balance is satisfied. This is a very crucial requirement for the correct calculation of stability: If the pressure balance is not satisfied up to high precision, there is not much for PB3D to do.
calculation of `DU`	Check how well the poloidal derivative of the geodesical component of the minimizing perturbation, U, is calculated by comparing it to the numerical derivative.

For the specific case of VMEC equilibria, finally, there is the additional `--plot_VMEC_modes` option that produces outputs of the harmonics of the different quantities: R, Z, λ, $\mathcal{J}$ and $\vec{B}$.

It is very important that these are well-converged, as the internal consistency depends strongly on this. Therefore, if this is not the case, be prepared to receive unaccurate stability results.

6.3 Setting Up the Input File

Now that we have the equilibrium files, it is time to have a look at the input file.

```
&inputdata_PB3D
    min_n_par_X             = 500
    min_par_X               = 0.0
    max_par_X               = 2.0
    alpha                   = 0.0
    min_r_sol               = 0.1
    max_r_sol               = 1.0
    n_r_sol                 = 500
    prim_X                  = 10
    n_mod_X                 = 20
    BC_style                = 1 1
    U_style                 = 3
    max_tot_mem             = 8000
    max_it_rich             = 2
    use_normalization       = .true.
/
//! [some extra options for PB3D]
    rich_restart_lvl        = 1
    tol_rich                = 1.0E-4
    tol_SLEPC               = 1.0E-8 1.0E-8
//! [some extra options for PB3D]
```

Have a look:

- `min_n_par_X` is the number of points in the parallel grid.
- They range from `min_par_X` to `max_par_X`. As this is an axisymmetric example, they can simply be chosen to be a fundamental period of 2π. For 3-D equilibria, this number has to be increased until convergence is reached [17].
- `alpha` indicates the field line at which the perturbations are situated.
- `n_r_sol` is the number of points in the solution grid, that ranges from `min_r_sol` to `max_r_sol` in the PB3D normal coordinate (see num_vars.use_pol_flux_f).
- `n_mod_X` is the number of resonating Harmonics with `prim_X` the primary mode number.
- `BC_style(1:2)` indicates the style with which the boundary conditions are implemented deep in the plasma and on the plasma edge.
- `max_tot_mem` is the total virtual memory that PB3D may consume.
- `use_normalization` indicates whether normalization is used (recommended), but it is useful to turn this off for debugging.

Note

`BC_style(2)` is currently set to 1, indicating that the mode amplitude is zero at the edge. In the upcoming version of PB3D, the vacuum perturbation on the potential energy will be taken into account, which enables other boundary condition styles at the plasma edge.

There are also some extra options listed (but not loaded in PB3D as they com after the end of the `inputdata_PB3D` block of the Fortran namelist) below:

```
rich_restart_lvl        = 1
tol_rich                = 1.0E-4
tol_SLEPC               = 1.0E-8 1.0E-8
```

They include:

- `rich_restart_lvl` is used to restart a simulation at a certain Richardson level. The Richardson level 1 refers to the first simulation, with the number of parallel points used in the calculation of line integrals along the magnetic field set by `min_n_par_X` (see 2.1). For every subsequent Richardson level, this number is approximately doubled in order to gain a grid size that is twice as fine. The results from these different levels are extrapolated to a theoretically infinitely fine grid. These operations are done in module rich_ops.
- `tol_rich` is the tolerance at which Richardson extrapolation is considered to have converged.
- `tol_SLEPC` is an array of tolerances used for the different SLEPc eigenvalue solutions.

There are many, many more input parameters. A short description of these can be found in Input variables.

6.4 Running PB3D

This tutorial provides a basic VMEC equilibrium file to experiment with, called `cbm18`, which represents a simple circular test case designed to be ballooning unstable. [5] Some properties are given in table 6.2.

Table 6.2: Table 1. equilibrium parameters

toroidal current I_{tot}	$1.614MA$

β_{pol}	0.443
aspect ratio ϵ	1.5

Put this equilibrium file, as well as the input file in a separate run directory; let's call it `cbm18`.

In order to produce outputs, PB3D needs three sub-directories in the run directory, called `Data/`, `Scripts/` and `Plots/`.

- In `Data/`, datafiles are stored that are used for plots.
- In `Scripts/`, scripts are generated that use the datafiles from `Data`.
- The `Plots/`, the resulting plots are located, both for external plots, and for HDF5 plots that can be read later in visualization software such as VisIt or ParaView. In Output Plots, this is explained in more detail.

If these directories are not present, you will receive error message, so go ahead and create them.

We can now run the code using `mpirun`. For example, with

```
mpirun -np 4 ./PB3D PB3D.input wout.nc
```

4 processes are used. You can find the VMEC file here.

There are optional run-time options that can be triggered, such as `--jump_to_sol`, which can be used to easily change solution preferences, such as discretization order, etc. For an overview, see Command-line inputs; they are not treated here.

You will see formatted output on the screen, with indentation that consistent indentation according to the depth of a routine in the program. The same output is also written to an output file, unformatted but still indented (see Log file).

The resulting eigenvalues can also be read in a separate output file (see Eigenvalue summary file).

Take a look at these output files and shuffle a bit with the input parameters if you want. For example, change `n_mod_X` to another value in order to get a different number Fourier harmonics, or `n_r_sol` to change the number of normal grid points in the solution. The solution should not change a lot if you have enough harmonics and normal grid poins.

Note

For VMEC, the magnetic axis is a bit problematic. It is necessary, therefore, to choose `min_r_sol` to be slightly larger than 0; 0.01 for example.

6.5 Run Script Tools

PB3D comes with a bunch of extra run tools, which can greatly improve user experience.

Note

These tools now come in a separate git repository, `PB3D_tools`.

In table 6.3, an overview is given:

Table 6.3: Table 2. extra tools

`run`	Run script that can be used for PB3D and POST with an extensive range of features including • the possibility to do a parameter scan for a variety of input variables • automatic support for schedulers such as PBS and SLURM • support for running on local folders when on a HPC cluster • etc... Note Use of this runscript is recommended, as it will make your life *much* easier, especially when you are working on a cluster.
`extract_jobs_info`	Extracts the eigenvalues from the results of a parameter scan produced by `run` with input variable modifications.
`inspect_SLURM_jobs`	Finds the directories of all the jobs currently running in SLURM and in every directory, it gives the tail of `PB3D.out`. The user can decide to cancel the job.

6.6 Post-processing With POST

POST provides a way to visualize the results from PB3D. Many options are possible, and they can be activated using `plot_` flags (see Input file).

An example is given in `POST.input`

```
&INPUTDATA_POST
    plot_resonance          = .true.
    plot_magn_grid          = .true.
    plot_flux_q             = .true.
    plot_sol_xi             = .true.
    plot_sol_Q              = .true.
    plot_E_rec              = .true.
    plot_B                  = .true.
    plot_J                  = .true.
    plot_kappa              = .true.
    min_r_plot              = 0.0
    max_r_plot              = 1.0
    n_theta_plot            = 101
    min_theta_plot          = 0.00
    max_theta_plot          = 2.00
    n_zeta_plot             = 1
    min_zeta_plot           = 0.00
    max_zeta_plot           = 0.00
    POST_style              = 1
    plot_grid_style         = 0
```

```
    pert_mult_factor_POST   = 0.0000
    max_tot_mem             = 8000
/
//! [some extra options for POST]
    ex_plot_style           = 2
    PB3D_rich_lvl           = 1
//! [some extra options for POST]
```

It can be seen that apart from the `plot_` flags, there are also

- `min_r_plot` and `max_r_plot`, which are the direct equivalent of `min_r_sol` and `max_r_sol` in PB3D.
- `min_theta_plot`, `max_theta_plot` and `n_theta_plot` which indicate the range of the poloidal variable that is plot, as well as the the number of point. There is an equivalent for the toroidal variable.
- `POST_style` that allows the user to change between different styles, such as whether the output grid is chosen to be fill the torus in a regular way, for example, or whether it follows the PB3D equilibrium grid.
- `plot_grid_style`, subsequently, can be used to change the way in which these grids are produced. For example, they could be changed to slab plots or plots with a straightened toroidal angle.
- `pert_mult_factor_POST` can be used to deform the output grids by the perturbation itself.

Also here, there are also some extra options listed below:

```
    ex_plot_style           = 2
    PB3D_rich_lvl           = 1
```

They include:

- `ex_plot_style` to indicate which program is to be used for external plotting. Changing it to 2 would engage Bokeh and Mayavi.
- `PB3D_rich_lvl` can be optionally employed to specify explicitely which Richardson level to use for the post-processing.

All of these options are explained in Input file. Again, there are also run-time options that can be found in Command-line inputs.

Now, run the POST program, for example using 4 MPI processes:

```
mpirun -np 4 ./POST POST.input PB3D_out.h5
```

An interesting output is the energy reconstruction, which is activated with `plot_E_rec`. This calculation reinserts the solution normal mode into the perturbed plasma potential terms, in order to check whether the Rayleigh quotient still holds, and to see which terms are dominant and which ones are less important. The results are stored in a separate files as well (see Energy reconstruction file).

If `--do_execute_command_line` was not explicitely used, the actual external plot programs are not used yet, but their usage is given in a shell commands script file (see Shell commands file). Run this shell with

```
./POST_shell_commands.sh
```

If everything is okay, you should now see output in the `Plots/` folder. If an error occurs, please verfiy whether you have appropriate versions of the external plot programs (i.e. GnuPlot, Bokeh, ...)

Also, open some of the `.xmf` files in VisIt or ParaView to explore these plots.

To finish this tutorial, feel free to adjust some input parameters. For example, you can change the output grid to become 3-D by setting `n_zeta_plot` and `max_zeta_plot`, or you could deform it using `pert_mult_factor_POST`.

Note

1. The cylindrical angle φ is often the inverse of the one used in nuclear fusion research, as it can lead to left-handed coordinate systems. In VMEC, it is opted to use it anyway, so it has a left-handed coordinate system. The poloidal angle is chosen to be anticlockwise if you look at a cross-section of the plasma that lies to the right of the Z-axis with the Z-axis pointing up, as in PB3D. In PB3D, however, the toroidal angle $\zeta = -\varphi$ is chosen as the inverse of the cylindrical angle, so it is right-handed.

Chapter 7

Installation

7.1 Introduction

PB3D is written in Fortran, and makes use of multiple numerical libraries:

- `blas / lapack`
  - for basic linear algebra
- `pblas / blacs / scalapack`
  - for parallelized basic linear algebra
- `HDF5`
  - for storage files
  - works in parallel
- `NetCDF`
  - to read input of VMEC
  - sequential
- `PETSc` / `SLEPc`
  - for linear algebra of large, sparse matrices
  - can reach $\sim \mathcal{O}(n)$ complexity
  - Minimal installation instructions:
    1. Configure PETSc using `./configure PETSC_ARCH=complex COPTFLAGS=-O3 CXXOPTFLAGS=-O3 FOPTFLAGS=-O3 \\endiskip`
       `--with-scalar-type=complex \\endiskip`
       `--download-metis \\endiskip`
       `--download-mumps \\endiskip`
       `--download-parmetis \\endiskip`
       `--with-scalapack-lib="-L[SCALAPACKDIR]/lib -lscalapack" \\endiskip`
       `--with-valgrind-dir=/usr \\endiskip`
       `--with-debugging=no \\endiskip`
       `--with-fortran-kernels=1`
       (If you want debug, remove `--with-debugging=no`, `--with-fortran-kernels=1`, COPTSFLAGS, CXXOPTFLAGS and FOPTFLAGS, and change PETSC_ARCH to `debug-complex`.)
       (If you use a different HDF5, add `--with-hdf5-dir=[HDF5DIR]`.)

2. follow instructions to do makes and tests.
3. Set global variables `export SLEPC_DIR=[SLEPC DIR]`, `export PETSC_DIR=[PETSC_DIR]` and `export PETSC_ARCH=[debug-]complex` (depending on whether it is debug or not).
4. Configure SLEPc using `./configure`
5. follow instructions to do makes and tests

- `StrumPack`
  - for linear algebra of structured matrices [1]
  - the vacuum potential energy perturbation calculations use such matrices as they are generated with a $1/r$ kernel.
  - can reach $\sim \mathcal{O}(n \log(n))$ complexity
  - we use the dense version of the package, which for Fortran is 1.1.1.
  - Minimal installation instructions:
    1. `Download`, untar and go to root folder.
    2. Go to examples, copy `Makefile.gnu` to `Makefile.inc` and adapt:
       (a) for example, it might be necessary to add `-lblacs` before `-lscalapack`
       (b) if you use intel, have a look at the `link line advisor`, and update the `LIB`
       (c) possibly, you might have to set the `CC` and `FC` compiler commands to, for example, `mpicc` respectively `mpiifort`
    3. Create directories `lib` and `inc`.
    4. Run `make f90_example`
    5. Make a library of `StrumpackDensePackage.o` using `ar -rcs libstrumpack.a *.o` in src.
    6. In inc, create a symlink `libstrumpack.a` to `libstrumpack.a` in src.
    7. In lib, create a symlink `strumpackdensepackage.mod` to `strumpackdensepackage.mod` in examples.

- `libstell`
  - part of Stellopt suite, which contains VMEC
  - provides routines to read VMEC output data

- `pspline`
  - Princeton Spline and Hermite Cubic Interpolation Routines
  - Minimal installation instructions:
    1. Export `FFLAGS` en `CFLAGS` if you want to optimize:
       * copy `share/Make.overwrite.sample` to `share/Make.overwrite`
       * Run the gmake below and note the flags, then edit these and put them in `share/Make.overwrite`
       * if GCC:
         - `FFLAGS = -c -O3 -m64 -fno-range-check -fdollar-ok -cpp ; export FFLAGS`
         - `CFLAGS = -c -O3 -m64 ; export CFLAGS`
         - `CXXFLAGS = -c -O3 -m64 ; export CXXFLAGS`
       * if Intel:
         - `FFLAGS = -c -O3 -nowarn -ftz -auto-scalar -traceback -align dcommons`
         - `MPI_FFLAGS = -c -O3 -nowarn -ftz -traceback -align dcommons -auto-scalar` (possibly add FC=mpiifort, `CXX=mpicpc`, `CC=mpiicc` if on a cluster)
    2. Compile with `gmake NETCDF_DIR= FORTRAN_VARIANT=[VARIANT]` with `[VARIANT]` either `GCC` or `Intel` (add `DEBUG=1` and remove the FLAGS if debug wanted). Possibly also add `OBJ=$[COMP_DIR]` if on a cluster, with `[COMP_DIR]` the directory where to put the resulting library.
    3. Run tests from `README_Pspline`.

They should probably be installed in this order. On linux distributions such as Ubuntu, they may be available as packages.

Furthermore, PB3D comes bundled with some other, smaller libraries:

- `fftpack`
  - to calculate the fast Fourier transform
- `foul`
  - the Fortran Output Library

These do not have to be installed separately.

7.2 Compilation

When all dependencies are satisfied, the program is then compiled in the standard way:

- Including the headers of all the libraries in the compilation of the object files:
  - This is done using `-I[path_to_library]`.
  - Make sure you add the `-o` option to create only object files.
- Linking with the actual libraries
  - This is done using `-L[path_to_library] -l[library_name]`.

7.3 Makefile Example

```
00001 ################################################################################
00002 #
00003 #   Example makefile for the program PB3D (Peeling Ballooning in 3D)
00004 #   \author Author: Toon Weyens
00005 #
00006 #   Don't forget to set the directories:
00007 #       - LIBSTELL_DIR
00008 #       - HDF5_DIR
00009 #       - NETCDFF_DIR (note: Fortran library)
00010 #       - PETSC_DIR
00011 #       - SLEPC_DIR
00012 #       - STRUMPACK_DIR
00013 ################################################################################
00014
00015 ################################################################################
00016 #   Include
00017 ################################################################################
00018 ## [PETSc and SLEPc trick]
00019 include  $(PETSC_DIR)/lib/petsc/conf/variables
00020 include  $(SLEPC_DIR)/lib/slepc/conf/slepc_variables
00021 ## [PETSc and SLEPc trick]
00022
00023 ## [PETSc and SLEPc trick inc]
00024 INCLUDE = $(PETSC_FC_INCLUDES) $(SLEPC_INCLUDE)
00025 ## [PETSc and SLEPc trick inc]
00026 ## [Libstell special]
00027 INCLUDE += -I$(LIBSTELL_DIR)/libstell_dir
00028 ## [Libstell special]
00029 INCLUDE += -I$(STRUMPACK_DIR)/include
00030 ## [PB3D include]
00031 INCLUDE += -I$(PB3D_DIR)/include
00032 ## [PB3D include]
00033 INCLUDE += -I/usr/include/hdf5/openmpi
00034
```

```
00035 ################################################################################
00036 #   Link
00037 ################################################################################
00038 ## [PB3D libraries]
00039 LIB_INTERNAL = libdfftpack.a libfoul.a libbspline.a
00040 ## [PB3D libraries]
00041
00042 LINK := $(LIB_INTERNAL)
00043
00044 ## [PETSc and SLEPc trick lib]
00045 LINK += $(PETSC_LIB)
00046 LINK += $(SLEPC_LIB)
00047 ## [PETSc and SLEPc trick lib]
00048 LINK += $(LIBSTELL_DIR)/libstell.a
00049 LINK += -L$(STRUMPACK_DIR)/lib -lstrumpack
00050 LINK += -L$(HDF5_DIR) -lhdf5_fortran -lhdf5
00051 LINK += -L$(NETCDFF_DIR)/lib -lnetcdff
00052 LINK += -Wl,-R$(NETCDFF_DIR)/lib
00053 LINK += -lscalapack -lblacs -lblas -lm
00054 LINK += -lstdc++ -lmpi_cxx
00055
00056
00057 ################################################################################
00058 #   Compiler
00059 ################################################################################
00060 COMPILER=mpifort
00061
00062
00063 ################################################################################
00064 #   Linker
00065 ################################################################################
00066 LINKER=mpifort
00067
00068
00069 ################################################################################
00070 #   Compiler flags
00071 #   options (used with -D[name]):
00072 #       ldebug: debug
00073 #       lIB: infiniband
00074 #       lwith_gnu: use GNU compiler [default]
00075 #       lwith_intel: use INTEL compiler, (checked for version 12.0.2)
00076 #   note: INTEL warning 6536 is suppressed, which informs about extra "USE".
00077 #   note: INTEL warning 6843 is suppressed, which informs about empty
00078 #       intent(out) variables
00079 ################################################################################
00080 COMP_FLAGS = -finit-real=snan -g -Og -Wall -Wextra -pedantic \
00081     -fimplicit-none -fbacktrace -fno-omit-frame-pointer \
00082     -fcheck=all -cpp -Dldebug# debug, profiling with gprof2dot, GCC
00083 #COMP_FLAGS = -O3 -fbacktrace -g -fimplicit-none -fno-omit-frame-pointer \
00084     #-cpp# optimized, GCC
00085
00086 #COMP_FLAGS = -O0 -DlIB -Dldebug -g -heap-arrays 100 -recursive \
00087     #-ftrapuv -check bounds -check uninit -traceback -implicitnone \
00088     #-fno-omit-frame-pointer -cpp -Dlwith_intel -diag-disable 6536 \
00089     #-diag-disable 6843# debug, profiling with gprof2dot, INTEL
00090 #COMP_FLAGS = -O3 -DlIB -traceback -g -heap-arrays 100 -recursive \
00091     #-implicitnone -fno-omit-frame-pointer -cpp -Dlwith_intel \
00092     #-diag-disable 6536 -diag-disable 6843# optimized, INTEL
00093
00094 COMP_FLAGS_EX= -O2 -w
00095
00096 COMP_FLAGS_F= -O2 -funroll-loops -fexpensive-optimizations
00097
00098
00099 ################################################################################
00100 #   Link flags
00101 ################################################################################
00102 LINK_FLAGS = -fPIC -finit-real=snan# debug
00103 #LINK_FLAGS = -fPIC# optimized
00104
00105
00106 ################################################################################
00107 #   Prepare
00108 ################################################################################
00109 # Add "Modules" and "Libraries" to the search path for the prerequisites
00110 VPATH = Modules:Libraries
00111
00112 # Contains list of source files (.o) and dependencies
00113 DEPLIST = PB3D.dep
00114 OBJLIST = ObjectList# defines "ObjectFiles"
00115
00116 # Includes source files and dependency list
00117 include $(DEPLIST)# Dependencies of all the objects
00118 include $(OBJLIST)# Names of all the objects
00119
00120
00121 ################################################################################
```

```
#   Rules
##############################################################################
all:    PB3D POST

PB3D:   $(ObjectFiles) $(LIB_INTERNAL) PB3D.o
    $(LINKER) -o $@ $(ObjectFiles) PB3D.o $(LINK) $(LINK_FLAGS)

POST:   $(ObjectFiles) $(LIB_INTERNAL) POST.o
    $(LINKER) -o $@ $(ObjectFiles) POST.o $(LINK) $(LINK_FLAGS)

libdfftpack.a:  dfft.o
    ar -rcs libdfftpack.a dfft.o

libfoul.a:  foul.o
    ar -rcs libfoul.a foul.o

libbspline.a:   bspline_sub_module.o
    ar -rcs libbspline.a bspline_sub_module.o

%.o: %.f90
    $(COMPILER) $(INCLUDE) $(COMP_FLAGS) -c $<

%.o: %.f
    $(COMPILER) $(COMP_FLAGS_F) -c $<

dfft.o: dfft.f
    $(COMPILER) $(COMP_FLAGS_EX) -c $<

foul.o: foul.f90
    $(COMPILER) $(COMP_FLAGS_EX) -c $<

bspline_sub_module.o: bspline_sub_module.f90
    $(COMPILER) $(COMP_FLAGS_EX) -c $<

clean:
    @rm -f *.o *.a *.mod *~ fort.*

clean_all:
    @rm -f *.o *.mod *~ fort.* PB3D POST
```

Note

1. PETSc and SLEPc don't like to be included in another makefile. The trick is to include two files:

   ```
   include  $(PETSC_DIR)/lib/petsc/conf/variables
   include  $(SLEPC_DIR)/lib/slepc/conf/slepc_variables
   ```

   which will load the variables `PETSC_FC_INCLUDES` and `SLEPC_INCLUDE`, used in

   ```
   INCLUDE = $(PETSC_FC_INCLUDES) $(SLEPC_INCLUDE)
   ```

   as well as the variables `PETSC_LIB` and `SLEPC_LIB`, used in

   ```
   LINK += $(PETSC_LIB)
   LINK += $(SLEPC_LIB)
   ```

2. There are versions of libstell that do not use the standard convention. In this case you have to look for the `*.mod` files. In the example makefile this is done with

   ```
   INCLUDE += -I$(LIBSTELL_DIR)/libstell_dir
   ```

   instead of the standard `inc` directory.

3. In

   ```
   INCLUDE += -I$(PB3D_DIR)/include
   ```

   there are includefiles that contain macros and wrappers specifically for PB3D.

4. In

   ```
   LIB_INTERNAL = libdfftpack.a libfoul.a libbspline.a
   ```

   linking is done with external libraries that are bundled with PB3D.

Chapter 8

Todo List

Module vac_ops

The vacuum part of PB3D is still under construction and not usable yet.

Chapter 9

Directory Documentation

9.1 Documentation Directory Reference

9.2 /Users/toonweyens/Code/PB3D/Modules Directory Reference

Directory dependency graph for Modules:

Files

- file driver_eq.f90
- file driver_POST.f90
- file driver_sol.f90
- file driver_X.f90
- file dtorh.f90
- file eq_ops.f90
- file eq_utilities.f90
- file eq_vars.f90
- file files_ops.f90
- file files_utilities.f90
- file grid_ops.f90
- file grid_utilities.f90

- file grid_vars.f90
- file HDF5_ops.f90
- file HDF5_utilities.f90
- file HDF5_vars.f90
- file HELENA_ops.f90
- file HELENA_vars.f90
- file input_ops.f90
- file input_utilities.f90
- file messages.f90
- file MPI_ops.f90
- file MPI_utilities.f90
- file MPI_vars.f90
- file num_ops.f90
- file num_utilities.f90
- file num_vars.f90
- file output_ops.f90
- file PB3D_ops.f90
- file PB3D_utilities.f90
- file rich_ops.f90
- file rich_vars.f90
- file SLEPC_ops.f90
- file SLEPC_utilities.f90
- file sol_ops.f90
- file sol_utilities.f90
- file sol_vars.f90
- file str_utilities.f90
- file test.f90
- file vac_ops.f90
- file vac_utilities.f90
- file vac_vars.f90
- file VMEC_ops.f90
- file VMEC_utilities.f90
- file VMEC_vars.f90
- file X_ops.f90
- file X_utilities.f90
- file X_vars.f90

9.3 /Users/toonweyens/Code/PB3D Directory Reference

Directories

- directory Modules

Files

- file PB3D.f90
- file POST.f90

Appendices

Appendix A

Programs

A.1 /Users/toonweyens/Code/PB3D/POST.f90 Program Rerefence

program __post_f90__

Peeling Ballooning in 3D: postprocessing.

A.1.1 Function/Subroutine Documentation

A.1.1.1 __post_f90__()

```
program __post_f90__
```

Peeling Ballooning in 3D: postprocessing.

Author

Toon Weyens, Contact: weyenst@gmail.com

Version

2.41

Date

2012-2018 Copyright (C) 2019 Toon Weyens

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the License along with this program. If not, contact weyenst@gmail.com

See also

References: [16] [17]

Definition at line 39 of file POST.

Here is the call graph for this function:

A.2 /Users/toonweyens/Code/PB3D/PB3D.f90 Program Rerefence

program __pb3d_f90__

Peeling Ballooning in 3D.

A.2.1 Function/Subroutine Documentation

A.2.1.1 __pb3d_f90__()

```
program __pb3d_f90__
```

Peeling Ballooning in 3D.

Author

Toon Weyens, Contact: weyenst@gmail.com

Version

2.41

Date

2012-2020 Copyright (C) 2020Toon Weyens

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the License along with this program. If not, contact weyenst@gmail.com

See also

References: [16] [17]

Definition at line 39 of file PB3D.

Here is the call graph for this function:

Appendix B

Modules

B.1 driver_eq Module Reference

Driver of the equilibrium part of PB3D.

Functions/Subroutines

- integer function, public run_driver_eq (grid_eq_out, grid_eq_b_out, eq_1_out, eq_2_out, vac)
  Main driver of PB3D equilibrium part.

B.1.1 Detailed Description

Driver of the equilibrium part of PB3D.

B.1.2 Function/Subroutine Documentation

B.1.2.1 run_driver_eq()

```
integer function, public driver_eq::run_driver_eq (
                type(grid_type), intent(inout), target grid_eq_out,
                type(grid_type), intent(inout), pointer grid_eq_b_out,
                type(eq_1_type), intent(inout) eq_1_out,
                type(eq_2_type), intent(inout) eq_2_out,
                type(vac_type), intent(inout) vac)
```

Main driver of PB3D equilibrium part.

- sets up ([out] means for output):
  - `grid_eq` [out] (for HELENA, only first Richardson level)
  - `grid_eq_B` [out] (for VMEC, equal to grid_eq_out)

  - `eq_1` [out] (only first Richardson level)
  - `eq_2` [out] (for HELENA, only first Richardson level) where output means
  - on the equilibrium grid if `X_grid` style is 1,3 (no change).
  - on a redistributed grid if `X_grid_style` is 2
- writes to HDF5:
  - `grid_eq` (for HELENA, only first Richardson level)
  - `grid_eq_B` (for VMEC, equal to grid_eq)
  - `eq_1` (only first Richardson level)
  - `eq_2` (only for HELENA)
- deallocates:
  - `grid_eq` [out] before setting up
  - `grid_B_eq` [out] before setting up
  - `eq_2` [out] before setting up

Returns

ierr

Parameters

`in,out`	*grid_eq_out*	output equilibrium grid
`in,out`	*grid_eq_b_↩ out*	output field-aligned equilibrium grid
`in,out`	*eq_1_out*	flux equilibrium variables in output grid
`in,out`	*eq_2_out*	metric equilibrium variables in output grid
`in,out`	*vac*	vacuum variables

Definition at line 47 of file driver_eq.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.2 driver_post Module Reference

Main driver of PostProcessing of program Peeling Ballooning in 3D.

Functions/Subroutines

- integer function, public init_post ()
  Initializes the POST driver.
- integer function, public run_driver_post ()
  The main driver routine for postprocessing.
- subroutine, public stop_post ()
  Cleans up main driver for postprocessing.

B.2.1 Detailed Description

Main driver of PostProcessing of program Peeling Ballooning in 3D.

B.2.2 Function/Subroutine Documentation

B.2.2.1 init_post()

```
integer function, public driver_post::init_post
```

Initializes the POST driver.

- set up preliminary variables
  - global variables
  - read grids (full)
  - `eq_1` (full) and `n` & `m` (full)
- set up output grids:
  - `POST_style = 1`: extended grid
  - `POST_style = 2`: field-aligned grid
- clean up
- set up final variables
  - normal limits
  - read grids (divided), `eq_1` (divided) and `sol` (divided) variables
- 1-D output plots
  - resonance plot
  - flux quantities plot
  - magnetic grid plot
- prepare Eigenvalue plots

  - calculates resonant surfaces
  - plots Eigenvalues
  - finds stability ranges for all Eigenvalues to be plot
  - plots harmonics for every Eigenvalue
  - calculates the parallel ranges of the equilibrium jobs
- clean up

In the actual driver, more detailed plots are possibly made for all requested Eigenvalues if `full_output`.

Returns

ierr

Definition at line 76 of file driver_POST.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.2.2.2 run_driver_post()

```
integer function, public driver_post::run_driver_post
```

The main driver routine for postprocessing.

Note

The PB3D output is given on different grids for different styles of the equilibrium model:

- VMEC: field-aligned grid on which EV problem has been solved.
- HELENA: output grid on which equilibrium and metric variables are tabulated.

Furthermore, if Richardson extrapolation is used, the VMEC grids and variables are contained in multiple HDF5 variables. These variables are needed here both in a field-aligned and a plot grid.

A consequence of the above is that for VMEC the field-aligned output is already given, but the output on the plot grid has to be calculated from scratch, while for HELENA both outputs have to be interpolated from the output tables.

The general workflow is as follows:

- take a subset of the output grids for the current equilibrium job.
- for `POST_style = 1` (extended grid):
  - VMEC: recalculate variables
  - HEL: interpolate variables for `POST_style = 2` (B-aligned grid):
  - VMEC: read subset of variables
  - HEL: interpolate variables
- create helper variables
- create plots and outputs

Returns

ierr

Definition at line 541 of file driver_POST.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.2.2.3 stop_post()

```
subroutine, public driver_post::stop_post
```

Cleans up main driver for postprocessing.

Definition at line 815 of file driver_POST.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.3 driver_sol Module Reference

Driver of the solution part of PB3D.

Functions/Subroutines

- integer function, public run_driver_sol (grid_eq, grid_x, grid_sol, x, vac, sol)
  Main driver of PB3D solution part.
- integer function interp_v (mds_i, grid_i, x_i, mds_o, grid_o, x_o)
  Interpolate tensorial perturbation quantities in the third dimension.

B.3.1 Detailed Description

Driver of the solution part of PB3D.

B.3.2 Function/Subroutine Documentation

B.3.2.1 interp_v()

```
integer function driver_sol::interp_v (
                type(modes_type), intent(in), target mds_i,
                type(grid_type), intent(in) grid_i,
                type(x_2_type), intent(in), target x_i,
                type(modes_type), intent(in), target mds_o,
                type(grid_type), intent(in) grid_o,
                type(x_2_type), intent(inout), target x_o)
```

Interpolate tensorial perturbation quantities in the third dimension.

The input grid should not be divided, though the output grid can be.

The procedure considers all possible mode number combinations. For `X_style` 2 (fast), each secondary mode only lives in a certain normal range of the plasma. Therefore, each secondary mode pair also has a limited normal range, given by the overlap of the ranges of the members.

The interpolated mode number combinations have a normal range that might slightly differ from the input ranges, in which case extrapolation can be done if the method allows for it.

Note

If the input grid is too coarse, but the interpolated grid is not, it is in theory possible that there are mode numbers and therefore mode number pairs that exist in the input grid, though they do so in the interpolated grid. In this case, they can not be calculated and are currently set to zero. However, the current implementation of setup_modes() produces an error if less than the full range of mode numbers is accurately present.

Parameters

in	*mds_i*	general modes variables for input
in	*grid_i*	grid at which `X_i` is tabulated
in	*x_i*	tensorial perturbation variable on input grid
in	*mds_o*	general modes variables for output
in	*grid_o*	grid at which `X_o` is interpolated
in,out	*x_o*	interpolated tensorial perturbation variable

Definition at line 283 of file driver_sol.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.3.2.2 run_driver_sol()

```
integer function, public driver_sol::run_driver_sol (
                type(grid_type), intent(in) grid_eq,
                type(grid_type), intent(in), target grid_x,
                type(grid_type), intent(inout) grid_sol,
                type(x_2_type), intent(in) x,
                type(vac_type), intent(inout) vac,
                type(sol_type), intent(inout) sol)
```

Main driver of PB3D solution part.

- sets up:
  - `grid_sol` (only first Richardson level)
  - `sol`
- writes to HDF5:
  - `grid_sol` (only first Richardson level)
  - `sol`
- deallocates:
  - sol before setting up (but after guess)

Returns

ierr

Parameters

`in`	*grid_eq*	equilibrium grid
`in`	*grid_x*	perturbation grid
`in,out`	*grid_sol*	solution grid
`in`	*x*	integrated tensorial perturbation variables
`in,out`	*vac*	vacuum variables
`in,out`	*sol*	solution variables

Definition at line 41 of file driver_sol.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.4 driver_x Module Reference

Driver of the perturbation part of PB3D.

Functions/Subroutines

- integer function, public run_driver_x (grid_eq, grid_eq_b, grid_x, grid_x_b, eq_1, eq_2, x_1, x_2)
  Main driver of PB3D perturbation part.

B.4.1 Detailed Description

Driver of the perturbation part of PB3D.

B.4.2 Function/Subroutine Documentation

B.4.2.1 run_driver_x()

```
integer function, public driver_x::run_driver_x (
                type(grid_type), intent(in), target grid_eq,
                type(grid_type), intent(in), pointer grid_eq_b,
                type(grid_type), intent(inout), target grid_x,
                type(grid_type), intent(inout), pointer grid_x_b,
                type(eq_1_type), intent(in), target eq_1,
                type(eq_2_type), intent(inout), target eq_2,
                type(x_1_type), intent(inout) x_1,
                type(x_2_type), intent(inout) x_2)
```

Main driver of PB3D perturbation part.

- sets up:
  - [0] `grid_X` (for HELENA, only first Richardson level)
  - [0] `grid_X_B` (for VMEC, equal to grid_X_out)
  - [1] `X_1`
  - [2] `X_2`
- writes to HDF5:
  - [0] `grid_X` (for HELENA, only first Richardson level)
  - [0] `grid_X_B` (for VMEC, equal to grid_X)
  - [1] `X_1` (only for HELENA, only first Richardson level)
  - [2] `X_2`
- deallocates:
  - `X_1` before setting up
  - `grid_X` before setting up
  - `grid_X_B` before setting up

([x] indicates driver x)

Note

eq_2 needs to be intent(inout) because interp_HEL_on_grid() requires this for generality. The variable is not modified in this driver, though.

Returns

ierr

Parameters

in	*grid_eq*	equilibrium grid (should be in but needs inout for interp_HEL_on_grid)
in	*grid_eq↩ _b*	field-aligned equilibrium grid (should be in but needs inout for interp_HEL_on_↩ grid)
in,out	*grid_x*	perturbation grid
in,out	*grid_x_b*	field-aligned perturbation grid
in	*eq_1*	flux equilibrium variables
in,out	*eq_2*	metric equilibrium variables (should be in but needs inout for interp_HEL_on_grid)
in,out	*x_1*	vectorial perturbation variables
in,out	*x_2*	tensorial perturbation variables

Definition at line 57 of file driver_X.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.5 dtorh Module Reference

Calculation of toroidal functions $P^m_{n-1/2}(z)$ and $Q^m_{n-1/2}(z)$.

Functions/Subroutines

- integer function, public dtorh1 (z, m, nmax, pl, ql, newn, mode, ipre)
  Calculates toroidal harmonics of a fixed order `m` and degrees up to `nmax`.

B.5.1 Detailed Description

Calculation of toroidal functions $P^m_{n-1/2}(z)$ and $Q^m_{n-1/2}(z)$.

Note

Copied and adapted from the DTORH1 routine by Segura and Gil [14] .

B.5.2 Function/Subroutine Documentation

B.5.2.1 dtorh1()

```
integer function, public dtorh::dtorh1 (
                real(dp), intent(in) z,
                integer, intent(in) m,
                integer, intent(in) nmax,
                real(dp), dimension(0:nmax), intent(inout) pl,
                real(dp), dimension(0:nmax), intent(inout) ql,
                integer, intent(inout) newn,
                integer, intent(in), optional mode,
                integer, intent(in), optional ipre)
```

Calculates toroidal harmonics of a fixed order `m` and degrees up to `nmax`.

Optionally, the `mode` can be specified to be different from 0 [default]

- if `mode=1`:
  - The set of functions evaluated is:
    $$\frac{P^m_{n-1/2}(z)}{\Gamma(m+1/2)}, \quad \frac{Q^m_{n-1/2}(z)}{\Gamma(m+1/2)},$$
    which are respectively stored in the arrays `pl(n)`, `ql(n)`.
  - newn refers to this new set of functions.
  - Note 1. and note 2. also apply in this case.
- if `mode=2`:
  - The code performs as for mode 1, but the restriction $z < 20$.
  - For the evaluation of the continued fraction is not considered.

Also, the parameter `ipre` can be used to select a different precision:

- For `ipre=1`, the precision is 10^{-12}, taking $\epsilon < 10^{-12}$
- For `ipre=2`, the precision is 10^{-8}, taking $\epsilon < 10^{-8}$

where ϵ controls the accuracy.

Warning

Use `mode` 2 only if high `m` 's for $z > 20$ are required. The evaluation of the cf may fail to converge for too high z 's.

Note

1. For a precision of 10^{-12}, if $z > 5$ and $z/m > 0.22$, the code uses a series expansion for `pl(0)`. When $z < 20$ and $z/m < 0.22$ a continued fraction is applied.
2. For a precision of 10^{-8}, if $z > 5$ and $z/m > 0.12$, the code uses a series expansion for `pl(0)`. When $z < 20$ and $z/m < 0.12$ a continued fraction is applied.

Returns

ierr

Parameters

in	*z*	(real) point at which toroidal harmonics are evaluated
in	*m*	order of toroidal harmonics (`m`>`0`)
in	*nmax*	maximum degree of toroidal harmonics (`nmax`>`0`)
in,out	*pl*	toroidal harmonics of first kind $P^m_{n-1/2}(z)$
in,out	*ql*	toroidal harmonics of second kind $Q^m_{n-1/2}(z)$
in,out	*newn*	maximum order of functions calculated when `pl(nmax+1)`>`1/tiny` with `tiny`= 10^{-290}
in	*mode*	mode that controls output
in	*ipre*	precision

Definition at line 74 of file dtorh.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.6 eq_ops Module Reference

Operations on the equilibrium variables.

Interfaces and Types

- interface calc_eq

  Calculate the equilibrium quantities on a grid determined by straight field lines.
- interface print_output_eq

  Print equilibrium quantities to an output file:
- interface redistribute_output_eq

  Redistribute the equilibrium variables, but only the Flux variables are saved.
- interface calc_rzl

  Calculate R, Z & λ and derivatives in VMEC coordinates.
- interface calc_g_c

  Calculate the lower metric elements in the C(ylindrical) coordinate system.
- interface calc_g_v

  Calculate the lower metric coefficients in the equilibrium V(MEC) coordinate system.
- interface calc_g_h

  Calculate the lower metric coefficients in the equilibrium H(ELENA) coordinate system.
- interface calc_g_f

  Calculate the metric coefficients in the F(lux) coordinate system.
- interface calc_jac_v

  Calculate $\mathcal{J}_V$, the jacobian in V(MEC) coordinates.
- interface calc_jac_h

  Calculate $\mathcal{J}_H$, the jacobian in HELENA coordinates.
- interface calc_jac_f

  Calculate $\mathcal{J}_F$, the jacobian in Flux coordinates.
- interface calc_t_vc

  Calculate $\overline{\overline{T}}_C^V$, the transformation matrix between C(ylindrical) and V(mec) coordinate system.
- interface calc_t_vf

  Calculate $\overline{\overline{T}}_V^F$, the transformation matrix between V(MEC) and F(lux) coordinate systems.
- interface calc_t_hf

  Calculate $\overline{\overline{T}}_H^F$, the transformation matrix between H(ELENA) and F(lux) coordinate systems.

Functions/Subroutines

- integer function, public flux_q_plot (grid_eq, eq)

  Plots the flux quantities in the solution grid.
- integer function, public calc_derived_q (grid_eq, eq_1, eq_2)

  Calculates derived equilibrium quantities system.
- integer function, public calc_normalization_const ()

  Sets up normalization constants.
- subroutine, public normalize_input ()

  Normalize input quantities.
- integer function, public b_plot (grid_eq, eq_1, eq_2, rich_lvl, plot_fluxes, xyz)

  Plots the magnetic fields.
- integer function, public j_plot (grid_eq, eq_1, eq_2, rich_lvl, plot_fluxes, xyz)

Plots the current.

- integer function, public kappa_plot (grid_eq, eq_1, eq_2, rich_lvl, xyz)

  Plots the curvature.
- integer function, public delta_r_plot (grid_eq, eq_1, eq_2, xyz, rich_lvl)

  *Plots **HALF** of the change in the position vectors for 2 different toroidal positions, which can correspond to a ripple.*
- integer function, public divide_eq_jobs (n_par_x, arr_size, n_div, n_div_max, n_par_x_base, range_name)

  Divides the equilibrium jobs.
- integer function, public calc_eq_jobs_lims (n_par_x, n_div)

  Calculate `eq_jobs_lims`.

Variables

- logical, public debug_calc_derived_q = .false.

  plot debug information for calc_derived_q()
- logical, public debug_j_plot = .false.

  plot debug information for j_plot()
- logical, public debug_create_vmec_input = .false.

  plot debug information for create_vmec_input()

B.6.1 Detailed Description

Operations on the equilibrium variables.

B.6.2 Function/Subroutine Documentation

B.6.2.1 b_plot()

```
integer function, public eq_ops::b_plot (
                type(grid_type), intent(inout) grid_eq,
                type(eq_1_type), intent(in) eq_1,
                type(eq_2_type), intent(in) eq_2,
                integer, intent(in), optional rich_lvl,
                logical, intent(in), optional plot_fluxes,
                real(dp), dimension(:,:,:,:), intent(in), optional xyz)
```

Plots the magnetic fields.

If multiple equilibrium parallel jobs, every job does its piece, and the results are joined automatically by plot_HDF5.

The outputs are given in contra- and covariant components and magnitude in multiple coordinate systems, as indicated in calc_vec_comp().

The starting point is the fact that the magnetic field is given by

$$\vec{B} = \frac{\vec{e}_\theta}{\mathcal{J}},$$

in F coordinates. The F covariant components are therefore given by

$$B_i = \frac{g_{i3}}{\mathcal{J}} = \frac{\vec{e}_i \cdot \vec{e}_3}{\mathcal{J}},$$

and the only non-vanishing contravariant component is

$$B^3 = \frac{1}{\mathcal{J}}.$$

These are then all be transformed to the other coordinate systems.

Note

1. Vector plots for different Richardson levels can be combined to show the total grid by just plotting them all individually.
2. The metric factors and transformation matrices have to be allocated.

Returns

ierr

Parameters

in,out	*grid_eq*	equilibrium grid
in	*eq_1*	flux equilibrium variables
in	*eq_2*	metric equilibrium variables
in	*rich_lvl*	Richardson level
in	*plot_fluxes*	plot the fluxes
in	*xyz*	X, Y and Z of grid

Definition at line 5698 of file eq_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.6.2.2 calc_derived_q()

```
integer function, public eq_ops::calc_derived_q (
                type(grid_type), intent(in) grid_eq,
                type(eq_1_type), intent(in) eq_1,
                type(eq_2_type), intent(inout), target eq_2)
```

Calculates derived equilibrium quantities system.

- magnetic shear `s`
- normal curvature `kappa_n`
- geodesic curvature `kappa_g`
- parallel current `sigma`

Naive implementations of these quantities, with the exception of `s`, give numerically very unfavorable results, so special care must be taken in this procedure.

This is an important issue as these derived equilbrium quantities are important building blocks of the perturbed potential energy, including the drivers of instabilities.

The general formulas are derived under the first section, VMEC. For axisymmetric HELENA, however, simplified equations are possible and these are presented afterwards.

B.6.3 VMEC

B.6.3.1 The shear

The local shear S is calculated using equation 3.22 from [15] :

$$S = -\frac{1}{\mathcal{J}}\frac{\partial}{\partial\theta}\left(\frac{h^{\psi\alpha}}{h^{\psi\psi}}\right)$$

which doesn't pose any particular problems as there are only angular derivatives.

B.6.3.2 The curvature

This is calculated using the parallel derivative of the parallel unit vector (i.e. theta if using poloidal flux and zeta if using toroidal flux).

By taking instead the derivative of the covariant basis vector and realizing that the difference with the real curvature lies solely in a component along the parallel direction, which cancels out when taking the normal or geodesic components, the situation becomes easier.

Asuming the poloidal flux is used as normal coordinate, the taks is therefore how to calculate $\frac{1}{|\vec{e}_\theta|^2}\frac{\partial}{\partial\theta}\vec{e}_\theta$.

By transforming both the derivative as well as the basis vector to Equilibrium coordinates, this can be written as

$\sum_{i=2,3}\sum_{j=2,3}\mathcal{T}^{\mathsf{E}}_{\mathsf{F}}(3,i))\frac{\partial}{\partial u^i_{\mathsf{E}}}\left(\mathcal{T}^{\mathsf{E}}_{\mathsf{F}}(3,j)\,\vec{e}_{j,\mathsf{E}}\right)$

where the summations only run over the angular coordinates because $\vec{e}_\theta$ never has any component along $\vec{e}_{\psi,\mathsf{E}}$.

This double summation formula can then be written as a matrix equation

$$\begin{pmatrix}\mathcal{T}^{\mathsf{E}}_{\mathsf{F}}(3,2) & \mathcal{T}^{\mathsf{E}}_{\mathsf{F}}(3,3)\end{pmatrix}\left[\begin{pmatrix}\frac{\partial}{\partial u^2}\vec{e}_2 & \frac{\partial}{\partial u^2}\vec{e}_3\\ \frac{\partial}{\partial u^3}\vec{e}_2 & \frac{\partial}{\partial u^3}\vec{e}_3\end{pmatrix}_{\mathsf{E}}\begin{pmatrix}\mathcal{T}^{\mathsf{E}}_{\mathsf{F}}(3,2)\\ \mathcal{T}^{\mathsf{E}}_{\mathsf{F}}(3,3)\end{pmatrix}+\begin{pmatrix}\frac{\partial}{\partial u^2}\mathcal{T}^{\mathsf{E}}_{\mathsf{F}}(3,2) & \frac{\partial}{\partial u^2}\mathcal{T}^{\mathsf{E}}_{\mathsf{F}}(3,3)\\ \frac{\partial}{\partial u^3}\mathcal{T}^{\mathsf{E}}_{\mathsf{F}}(3,2) & \frac{\partial}{\partial u^3}\mathcal{T}^{\mathsf{E}}_{\mathsf{F}}(3,3)\end{pmatrix}\begin{pmatrix}\vec{e}_2\\ \vec{e}_3\end{pmatrix}\right]$$

which can be represented shorthand as $\mathcal{T}^{\mathsf{E}}_{\mathsf{F}}(3,2:3)\left[\mathcal{D}\vec{e}_{\mathsf{E}}\mathcal{T}^{\mathsf{E}}_{\mathsf{F}}(3,2:3)^T+\mathcal{D}\mathcal{T}\vec{e}^T_{\mathsf{E}}\right]$

It is relatively easy to set up the matrix $\mathcal{D}\vec{e}_{\mathsf{E}}$ as a function of covariant basis vectors in the Cylindrical coordinate system. The derivatives of the transformation matrix itself can likewise be found.

The steps used in this routine are therefore

1. Set up $\mathcal{D}\vec{e}_{\mathsf{E}}$, i.e. as a function of the three cylindrical covariant basis vectors.
2. Set up correctcion by the derivatives of the transformation matrix, with a single summation.
3. Decompose the normal $\frac{\nabla\psi}{|\nabla\psi|^2}$ and geodesic vectors $\frac{\nabla\psi\times\vec{B}}{B^2}$ as a function of the cylindrical contravariant basis vectors.
4. Double summation to reduce term $\sim\mathcal{D}\vec{e}_{\mathsf{E}}$ and correction by single summation to reduce term $\sim\mathcal{D}\mathcal{T}$.
5. Dot these for each of the four (actually three because of symmetry) elements of $\mathcal{D}\vec{e}_{\mathsf{E}}$.
6. Divide by $|\vec{e}_\theta|^2$.
7. Possibly correct for toroidal flux.

An advantage of using this formulation is that no normal derivatives are needed, so that nothing has to implicitely cancel out.

B.6.3.3 The parallel current

The parallel current is calculated from the shear with the help of equation 3.29 of [15] :

$$\mu_0\sigma=-\frac{1}{B_\theta}\left[2\frac{\nabla\psi\times\vec{B}}{|\nabla\psi|^2}\cdot\frac{\partial(\nabla\psi)}{\partial\theta}+\mathcal{J}\,|\nabla\psi|^2\,S\right].$$

where a similar technique can be used as above, for the calculation of the curvature: As

$$\frac{\nabla\psi\times\vec{B}}{|\nabla\psi|^2}=\frac{B_\theta\vec{e}_\alpha-B_\alpha\vec{e}_\theta}{\mathcal{J}|\nabla\psi|^2},$$

The parallel derivative of $\nabla\psi$ can be rewritten in terms of contravariant components of derivatives of covariant basis vectors:

$$\begin{cases}\vec{e}_\alpha\cdot\dfrac{\partial(\nabla\psi)}{\partial\theta}=-\nabla\psi\cdot\dfrac{\partial\vec{e}_\theta}{\partial\alpha}\\ \vec{e}_\theta\cdot\dfrac{\partial(\nabla\psi)}{\partial\theta}=-\nabla\psi\cdot\dfrac{\partial\vec{e}_\theta}{\partial\theta}\end{cases},$$

so that the result is:

$$\frac{\nabla\psi\times\vec{B}}{|\nabla\psi|^2}\cdot\frac{\partial(\nabla\psi)}{\partial\theta}=\frac{1}{\mathcal{J}^2}\frac{\nabla\psi}{|\nabla\psi|^2}\cdot\left[g_{\alpha\theta}\frac{\partial\vec{e}_\theta}{\partial\theta}-g_{\theta\theta}\frac{\partial\vec{e}_\theta}{\partial\alpha}\right],$$

which is given by a formula similar to the one used above for the geodesical curvature.

In debug mode, it can be checked whether the current is indeed divergence-free, with the help of equation 3.33 of [15].

$$-2p'\int_{\theta_0}^{\theta}\mathcal{J}\kappa_g\mathrm{d}\theta=\sigma\left(\theta\right)-\sigma_0$$

and whether a direct, naive implementation of the parallel current from equation 3.26 of [15] agrees with the more accurate results used here:

$$\mu_0\sigma=\frac{\partial B_\psi}{\partial\alpha}-\frac{\partial B_\alpha}{\partial\psi}-\mu_0p'\mathcal{J}\frac{B_\alpha}{B_\theta}.$$

The reason why they are generally different is that this implementation relies on the cancellation of large terms.

B.6.4 HELENA

B.6.4.1 The parallel current

As $B_\alpha=F\left(\psi\right)$ and $\vec{e}_{\alpha,\mathrm{F}}=\vec{e}_{\phi,\mathrm{H}}$, the naive expression for the shear becomes simply

$$\sigma=-F'-\mu_0p'\frac{F}{B^2},$$

which can be used like that.

B.6.4.2 The shear

The calculate the local shear S is looks like it is best to use equation 3.22 from [15], just as in the VMEC case:

As a test, however, equation 3.29 of [15] can be used in stead:

$$\mathcal{J}\left|\nabla\psi\right|^2S+\mu_0\mathcal{J}B^2\sigma=-2\frac{F}{R}\left(\frac{Z_\theta}{R}+\frac{Z_\theta R_{\theta\theta}-R_\theta Z_{\theta\theta}}{R_\theta^2+Z_\theta^2}\right).$$

from which σ can be obtained.

B.6.4.3 The curvature

Also the curvature expressions can be simplified for axisymmetric situations. The result is given by

$$\kappa_n = \frac{qR}{F}\frac{Z_\theta(R_{\theta\theta}-q^2R)-R_\theta Z_{\theta\theta}}{\left(R_\theta^2+Z_\theta^2+(qR)^2\right)(R_\theta^2+Z_\theta^2)}$$

$$\kappa_g = qR\frac{R_\theta\left(2(R_\theta^2+Z_\theta^2)+(qR)^2\right)-R(R_\theta R_{\theta\theta}+Z_\theta Z_{\theta\theta})}{\left(R_\theta^2+Z_\theta^2+(qR)^2\right)^2}$$

Note

1. If the toroidal flux is used instead, the actual curvature obviously is unchanged, which implies that the normal component has to be multiplied by the safety factor and the geodesic component has to be divided by it.
2. The formulas for the normal and geodesic basis vectors for VMEC are
   - $\frac{\nabla\psi}{|\nabla\psi|^2} = -\frac{\mathcal{J}(1+\lambda_\theta)}{Z_\theta^2+R_\theta^2+(R_\zeta Z_\theta-R_\theta Z_\zeta)/R^2}\frac{1}{R}\left(-Z_\theta, R_\zeta Z_\theta - R_\theta Z_\zeta, R_\theta\right)_{\mathsf{C}}$
   - $\frac{\nabla\psi\times\vec{B}}{B^2} = \frac{1}{g_{\theta\theta}}\left(PR_\theta - QR_\zeta, -QR^2, PZ_\theta - QZ_\zeta\right)_{\mathsf{C}}$
   - $Q = g_{\theta\theta} - qg_{\theta\alpha}$ (using right-handed flux q_{F}, which is the inverse of the left-handed VMEC q_{V})
   - $P = \frac{Q\lambda_\zeta - g_{\alpha\theta}}{1+\lambda_\theta}$ (where the derivatives of R, Z and λ are in VMEC coordinates).
3. The formulas for the normal and geodesic basis vectors for HELENA are
   - $\frac{\nabla\psi}{|\nabla\psi|^2} = -\frac{1}{Z_\theta^2+R_\theta^2}\frac{qR}{F}\left(-Z_\theta, 0, R_\theta\right)_{\mathsf{C}}$
   - $\frac{\nabla\psi\times\vec{B}}{B^2} = \frac{qR^2}{R_\theta^2+Z_\theta^2+q^2R^2}\left(-R_\theta, -\frac{R_\theta^2+Z_\theta^2}{q}, -Z_\theta\right)_{\mathsf{C}}$

Parameters

in	*grid_eq*	equilibrium grid
in	*eq_1*	flux equilibrium variables
in,out	*eq_2*	metric equilibrium variables

Definition at line 4286 of file eq_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.6.4.4 calc_eq_jobs_lims()

```
integer function, public eq_ops::calc_eq_jobs_lims (
            integer, intent(in) n_par_x,
            integer, intent(in) n_div)
```

Calculate `eq_jobs_lims`.

Take into account that every job has to start and end at the start and end of a fundamental integration interval, as discussed in divide_eq_jobs():

- for `magn_int_style=1` (trapezoidal), this is 1 point,
- for `magn_int_style=2` (Simpson 3/8), this is 3 points.

for POST, there are no Richardson levels, and there has to be overlap of one always, in order to have correct composite integrals of the different regions.

Note

The `n_par_X` passed into this procedure refers to the quantity that is already possibly halved if the Richardson level is higher than 1. This information is then reflected in the eq_jobs_lims, which refer to the local limits, i.e. only the parallel points currently under consideration.

Returns

ierr

Parameters

in	*n_par_x*	number of parallel points in this Richardson level
in	*n_div*	nr. of divisions of parallel ranges

Definition at line 6700 of file eq_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.6.4.5 calc_normalization_const()

```
integer function, public eq_ops::calc_normalization_const
```

Sets up normalization constants.

- VMEC version (`eq_style=1`):
  Normalization depends on `norm_style`:
  1. MISHKA
     - R_0: major radius (= average magnetic axis)
     - B_0: B on magnetic axis (theta = zeta = 0)
     - pres_0: reference pressure (= B_0^2/mu_0)
     - psi_0: reference flux (= R_0^2 B_0)
     - rho_0: reference mass density
  2. COBRA
     - R_0: major radius (= average geometric axis)
     - pres_0: pressure on magnetic axis
     - B_0: reference magnetic field (= sqrt(2pres_0mu_0 / beta))
     - psi_0: reference flux (= R_0^2 B_0 / aspr^2)
     - rho_0: reference mass density where aspr (aspect ratio) and beta are given by VMEC.
- HELENA version (`eq_style=2`):
  MISHKA Normalization is used by default and does not depend on `norm_style`

See also

read_hel()

Note

`rho_0` is not given through by the equilibrium codes and should be user-supplied

Returns

ierr

Definition at line 5404 of file eq_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.6.4.6 delta_r_plot()

```
integer function, public eq_ops::delta_r_plot (
                type(grid_type), intent(inout) grid_eq,
                type(eq_1_type), intent(in) eq_1,
                type(eq_2_type), intent(in) eq_2,
                real(dp), dimension(:,:,:,:), intent(in) xyz,
                integer, intent(in), optional rich_lvl)
```

Plots **HALF** of the change in the position vectors for 2 different toroidal positions, which can correspond to a ripple.

Also calculates **HALF** of the relative magnetic perturbation, which also corresponds to a ripple.

Finally, if the output grid contains a fundamental interval 2π, the proportionality between both is written to a file.

Note

The metric factors and transformation matrices have to be allocated.

Returns

ierr

Parameters

in,out	*grid_eq*	equilibrium grid
in	*eq_1*	flux equilibrium variables
in	*eq_2*	metric equilibrium variables
in	*xyz*	X, Y and Z of grid
in	*rich_lvl*	Richardson level

Definition at line 6171 of file eq_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.6.4.7 divide_eq_jobs()

```
integer function, public eq_ops::divide_eq_jobs (
                integer, intent(in) n_par_x,
                integer, dimension(2), intent(in) arr_size,
                integer, intent(inout) n_div,
                integer, intent(in), optional n_div_max,
                integer, intent(in), optional n_par_x_base,
                character(len=*), intent(in), optional range_name)
```

Divides the equilibrium jobs.

For PB3D, the entire parallel range has to be calculated, but due to memory limits this has to be split up in pieces. Every piece has to be able to contain the equilibrium variables (see note below), as well as the vectorial perturbation variables. These are later combined into tensorial variables and integrated.

The equilibrium variables have to be operated on to calculate them, which translates to a scale factor `mem_scale_fac`. However, in the perturbation phase, when they are just used, this scale factor is not needed.

In its most extreme form, the division in equilibrium jobs would be the individual calculation on a fundamental integration integral of the parallel points:

- for `magn_int_style=1` (trapezoidal), this is 1 point,
- for `magn_int_style=2` (Simpson 3/8), this is 3 points.

For HELENA, the parallel derivatives are calculated discretely, the equilibrium and vectorial perturbation variables are tabulated first in this HELENA grid. This happens in the first Richardson level. In all Richardson levels, afterwards, these variables are interpolated in the angular directions. In this case, therefore, there can be no division of this HELENA output interval for the first Richardson level.

This procedure does the job of dividing the grids setting the global variables `eq_jobs_lims`.

The integration of the tensorial perturbation variables is adjusted:

- If the first job of the parallel jobs and not the first Richardson level: add half of the integrated tensorial perturbation quantities of the previous level.
- If not the first job of the parallel jobs, add the integrated tensorial perturbation quantities to those of the previous parallel job, same Richardson level.

In fact, the equilibrium jobs have much in common with the Richardson levels, as is attested by the existence of the routines do_eq() and eq_info(), which are equivalent to do_rich() and rich_info().

In POST, finally, the situation is slightly different for HELENA, as all the requested variables have to fit, including the interpolated variables, as they are stored whereas in PB3D they are not. The parallel range to be taken is then the one of the output grid, including a base range for the variables tabulated on the HELENA grid. Also, for extended output grids, the size of the grid in the secondary angle has to be included in `n_par_X` (i.e. toroidal when poloidal flux is used and vice versa). Furthermore, multiple equilibrium jobs are allowed.

To this end, optionally, a base number can be provided for `n_par_X`, that is always added to the number of points in the divided `n_par_X`.

Note

For PB3D, only the variables `g_FD`, `h_FD` and `jac_FD` are counted, as the equilibrium variables and the transformation matrices are deleted after use. Also, `S`, `sigma`, `kappa_n` and `kappa_g` can be neglected as they do not contain derivatives and are therefore much smaller. in both routines calc_memory_eq() and calc_memory_x(), however, a 50% safety factor is used to account for this somewhat.

Returns

ierr

Parameters

`in`	*n_par_x*	number of parallel points to be divided

in	*arr_size*	array size (using loc_n_r) for eq_2 and X_1 variables
in,out	*n_div*	final number of divisions
in	*n_div_max*	maximum n_div
in	*n_par_x_base*	base n_par_X, undivisible
in	*range_name*	name of range

Definition at line 6564 of file eq_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.6.4.8 flux_q_plot()

```
integer function, public eq_ops::flux_q_plot (
                type(grid_type), intent(in) grid_eq,
                type(eq_1_type), intent(in) eq)
```

Plots the flux quantities in the solution grid.

- safety factor `q_saf`
- rotational transform `rot_t`
- pressure `pres`
- poloidal flux `flux_p`
- toroidal flux `flux_t`

Returns

ierr

Parameters

in	*grid_eq*	normal grid
in	*eq*	flux equilibrium variables

Definition at line 3809 of file eq_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.6.4.9 j_plot()

```
integer function, public eq_ops::j_plot (
                type(grid_type), intent(inout) grid_eq,
                type(eq_1_type), intent(in) eq_1,
                type(eq_2_type), intent(in) eq_2,
                integer, intent(in), optional rich_lvl,
                logical, intent(in), optional plot_fluxes,
                real(dp), dimension(:,:,:,:), intent(in), optional xyz)
```

Plots the current.

If multiple equilibrium parallel jobs, every job does its piece, and the results are joined automatically by plot_HDF5.

The outputs are given in contra- and covariant components and magnitude in multiple coordinate systems, as indicated in calc_vec_comp().

The starting point is the pressure balance

$$\nabla p = \vec{J} \times \vec{B},$$

which, using $\vec{B} = \frac{\vec{e}_\theta}{\mathcal{J}}$, reduces to

$$J^\alpha = -p'.$$

Furthermore, the current has to lie in the magnetic flux surfaces:

$$J^\psi = 0.$$

Finally, the parallel current σ gives an expression for the last contravariant component:

$$J^\theta = \frac{\sigma}{\mathcal{J}} + p' \frac{B_\alpha}{B_\theta}.$$

From these, the contravariant components can be calculated as

$$J_i = J^\alpha g_{\alpha,i} + J^\theta g_{\theta,i}.$$

These are then all be transformed to the other coordinate systems.

Note

1. Vector plots for different Richardson levels can be combined to show the total grid by just plotting them all individually.
2. The metric factors and transformation matrices have to be allocated.

Returns

ierr

Parameters

`in,out`	*grid_eq*	equilibrium grid
`in`	*eq_1*	flux equilibrium variables
`in`	*eq_2*	metric equilibrium variables
`in`	*rich_lvl*	Richardson level
`in`	*plot_fluxes*	plot the fluxes
`in`	*xyz*	X, Y and Z of grid

Definition at line 5801 of file eq_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.6.4.10 kappa_plot()

```
integer function, public eq_ops::kappa_plot (
                type(grid_type), intent(inout) grid_eq,
                type(eq_1_type), intent(in) eq_1,
                type(eq_2_type), intent(in) eq_2,
                integer, intent(in), optional rich_lvl,
                real(dp), dimension(:,:,:,:), intent(in), optional xyz)
```

Plots the curvature.

If multiple equilibrium parallel jobs, every job does its piece, and the results are joined automatically by plot_HDF5.

The outputs are given in contra- and covariant components and magnitude in multiple coordinate systems, as indicated in calc_vec_comp().

The starting point is the curvature, given by

$$\vec{\kappa} = \kappa_n \frac{\nabla\psi}{|\nabla\psi|^2} + \kappa_g \frac{\nabla\psi \times \vec{B}}{B^2},$$

which can be used to find the covariant and contravariant components in Flux coordinates.

These are then transformed to Cartesian coordinates and plotted.

Note

1. Vector plots for different Richardson levels can be combined to show the total grid by just plotting them all individually.
2. The metric factors and transformation matrices have to be allocated.

Returns

ierr

Parameters

in,out	*grid_eq*	equilibrium grid
in	*eq_1*	metric equilibrium variables
in	*eq_2*	metric equilibrium variables
in	*rich_lvl*	Richardson level
in	*xyz*	X, Y and Z of grid

Definition at line 5969 of file eq_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.6.4.11 normalize_input()

```
subroutine, public eq_ops::normalize_input
```

Normalize input quantities.

See also

calc_normalization_const()

Definition at line 5642 of file eq_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.6.5 Variable Documentation

B.6.5.1 debug_calc_derived_q

```
logical, public eq_ops::debug_calc_derived_q = .false.
```

plot debug information for calc_derived_q()

Note

Debug version only

Definition at line 30 of file eq_ops.f90.

B.6.5.2 debug_create_vmec_input

```
logical, public eq_ops::debug_create_vmec_input = .false.
```

plot debug information for create_vmec_input()

Note

Debug version only

Definition at line 34 of file eq_ops.f90.

B.6.5.3 debug_j_plot

```
logical, public eq_ops::debug_j_plot = .false.
```

plot debug information for j_plot()

Note

Debug version only

Definition at line 32 of file eq_ops.f90.

B.7 eq_utilities Module Reference

Numerical utilities related to equilibrium variables.

Interfaces and Types

- interface calc_f_derivs

  Transforms derivatives of the equilibrium quantities in E coordinates to derivatives in the F coordinates.
- interface calc_inv_met

  Calculate $D_1^{m_1} D_2^{m_2} D_3^{m_3} X$ *from* $D_1^{i_1} D_2^{i_2} D_3^{i_3} X$ *and* $D_1^{j_1} D_2^{j_2} D_3^{j_3} Y$ *where* $XY = 1$ *and* $i, j = 0 \ldots m$*, according to [15].*
- interface transf_deriv

  Calculates derivatives in a coordinate system B from derivatives in a coordinates system A, making use of the transformation matrix $\overline{\overline{T}}_B^A$.

Functions/Subroutines

- integer function, public calc_g (g_a, t_ba, g_b, deriv, max_deriv)

  Calculate the metric coefficients in a coordinate system B ! using the.
- integer function, public calc_memory_eq (arr_size, n_par, mem_size)

  Calculate memory in MB necessary for variables in equilibrium job.
- logical function, public do_eq ()

  If this equilibrium job should be done, also increment `eq_job_nr`.
- elemental character(len=max_str_ln) function, public eq_info ()

  Returns string with possible extension with equilibrium job as well as parallel job, or nothing if only one level and one parallel job.
- subroutine, public print_info_eq (n_par_x_rich)

  Prints information for equilibrium parallel job.

Variables

- logical, public debug_calc_inv_met_ind = .false.

  plot debug information for calc_inv_met_ind()

B.7.1 Detailed Description

Numerical utilities related to equilibrium variables.

B.7.2 Function/Subroutine Documentation

B.7.2.1 calc_g()

```
integer function, public eq_utilities::calc_g (
                real(dp), dimension(:,:,:,:,0:,0:,0:), intent(in) g_a,
                real(dp), dimension(:,:,:,:,0:,0:,0:), intent(in) t_ba,
                real(dp), dimension(:,:,:,:,0:,0:,0:), intent(inout) g_b,
                integer, dimension(3), intent(in) deriv,
                integer, intent(in) max_deriv)
```

Calculate the metric coefficients in a coordinate system B ! using the.

Parameters

in	*g_a*	g_A
in	*t_ba*	$\overline{T}_B^A$
in,out	*g_b*	g_B
in	*deriv*	derivatives
in	*max_deriv*	maximum derivative

Definition at line 386 of file eq_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.7.2.2 calc_memory_eq()

```
integer function, public eq_utilities::calc_memory_eq (
                integer, intent(in) arr_size,
                integer, intent(in) n_par,
                real(dp), intent(inout) mem_size)
```

Calculate memory in MB necessary for variables in equilibrium job.

The size of these variables is equal to the product of the non-parallel dimensions (e.g. `n_geo` x `loc_n_r`), times the number of variables.

The latter should be:

- PB3D: only take into account ($2 \cdot 6 + 1 = 13$) equilibrium variables `g_FD`, `h_FD` and `jac_FD`, as the perturbation variables are divided in jobs occupying the remaning space. These equilibrium variables are tabulated on the equilibrium grid. Note that they contain derivatives in extra dimensions, so that their size should be multiplied by (`max_deriv`+1)^3.
- POST: take into account these 13 equilibrium variables, as well as 4 variables `U` and `DU`, with double size due to being complex, and the additional dimension equal to `n_mod_X`, but without derivatives.

Returns

ierr

Parameters

in	*arr_size*	size of part of X array
in	*n_par*	number of parallel points
in,out	*mem_size*	total size

Definition at line 906 of file eq_utilities.f90.

Here is the caller graph for this function:

B.7.2.3 do_eq()

```
logical function, public eq_utilities::do_eq
```

If this equilibrium job should be done, also increment `eq_job_nr`.

Definition at line 948 of file eq_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.7.2.4 eq_info()

```
elemental character(len=max_str_ln) function, public eq_utilities::eq_info
```

Returns string with possible extension with equilibrium job as well as parallel job, or nothing if only one level and one parallel job.

Definition at line 973 of file eq_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.7.2.5 print_info_eq()

```
subroutine, public eq_utilities::print_info_eq (
            integer, intent(in) n_par_x_rich)
```

Prints information for equilibrium parallel job.

Parameters

in	*n_par_x_rich*	number of parallel points in this Richardson level

Definition at line 985 of file eq_utilities.f90.

Here is the caller graph for this function:

B.7.3 Variable Documentation

B.7.3.1 debug_calc_inv_met_ind

```
logical, public eq_utilities::debug_calc_inv_met_ind = .false.
```

plot debug information for calc_inv_met_ind()

Note

Debug version only

Definition at line 27 of file eq_utilities.f90.

B.8 eq_vars Module Reference

Variables that have to do with equilibrium quantities and the grid used in the calculations:

Interfaces and Types

- type eq_1_type
  flux equilibrium type
- type eq_2_type
  metric equilibrium type

Functions/Subroutines

- subroutine init_eq_1 (eq, grid, setup_e, setup_f)

  Initializes new flux equilibrium.
- subroutine init_eq_2 (eq, grid, setup_e, setup_f)

  Initializes new metric equilibrium.
- subroutine copy_eq_1 (eq_i, grid_i, eq_o)

  Deep copy of flux equilibrium variables.
- subroutine copy_eq_2 (eq_i, grid_i, eq_o)

  Deep copy of metric equilibrium variables.
- subroutine dealloc_eq_1 (eq)

  Deallocates flux equilibrium quantities.
- subroutine dealloc_eq_2 (eq)

  Deallocates metric equilibrium quantities.

Variables

- real(dp), public r_0

  independent normalization constant for nondimensionalization
- real(dp), public pres_0

  independent normalization constant for nondimensionalization
- real(dp), public rho_0

  independent normalization constant for nondimensionalization
- real(dp), public b_0

  derived normalization constant for nondimensionalization
- real(dp), public psi_0

  derived normalization constant for nondimensionalization
- real(dp), public t_0

  derived normalization constant for nondimensionalization
- real(dp), public vac_perm = mu_0_original

  either usual mu_0 (default) or normalized
- real(dp), public max_flux_e

  max. flux in Equilibrium coordinates, set in calc_norm_range_PB3D_in
- real(dp), public max_flux_f

  max. flux in Flux coordinates, set in calc_norm_range_PB3D_in
- integer, public n_alloc_eq_1s

  nr. of allocated eq_1 variables
- integer, public n_alloc_eq_2s

  nr. of allocated eq_2 variables

B.8.1 Detailed Description

Variables that have to do with equilibrium quantities and the grid used in the calculations:

- The equilibrium variables are comprised of the variables that result from the equilibrium calculation, such as pressure, rotational transform, etc.
- The flux variables are tabulated on a 1D grid.

- The metric variables are tabulated on a 3D grid
  - In the normal coordinate, they are tabulated in the equilibrium grid.
  - In the angular coordinates, they are tabulated in the solution grid (VMEC), or in the equilibrium grid followed by an adaptation to the solution grid (HELENA).
- However, this is not necessarily always the case, as it depends on the grid angles being aligned with the grid (e.g. Providing theta and zeta in the equilibrium grid so that the magnetic field lines are followed.

Note

In general in PB3D, there are two kinds of variables, differing from one another in the way in which they are tabulated:

- variables tabulated on the full output grid of the equilibrium code
- variables tabulated in an internal grid of this code

In many places in the code a range in the normal coordinate is selected for each of the variables on different processes. This selection has to be done correctly and things can get a little bit complicated if trimmed grids are used (grids that have no overlap between processes).

See also

grid_ops()

B.8.2 Function/Subroutine Documentation

B.8.2.1 copy_eq_1()

```
subroutine eq_vars::copy_eq_1 (
                class(eq_1_type), intent(in) eq_i,
                type(grid_type), intent(in) grid_i,
                type(eq_1_type), intent(inout) eq_o)
```

Deep copy of flux equilibrium variables.

Parameters

in	*eq_i*	eq_1 to be copied
in	*grid↩ _i*	grid of eq_i
in,out	*eq_o*	copied eq_1

Definition at line 438 of file eq_vars.f90.

B.8.2.2 copy_eq_2()

```
subroutine eq_vars::copy_eq_2 (
                class(eq_2_type), intent(in) eq_i,
                type(grid_type), intent(in) grid_i,
                type(eq_2_type), intent(inout) eq_o)
```

Deep copy of metric equilibrium variables.

Returns

ierr

Parameters

in	*eq_i*	eq_2 to be copied
in	*grid↩ _i*	grid of eq_i
in,out	*eq_o*	copied eq_1

Definition at line 476 of file eq_vars.f90.

B.8.2.3 dealloc_eq_1()

```
subroutine eq_vars::dealloc_eq_1 (
                class(eq_1_type), intent(inout) eq)
```

Deallocates flux equilibrium quantities.

Parameters

in,out	*eq*	equilibrium to be deallocated

Definition at line 530 of file eq_vars.f90.

Here is the call graph for this function:

B.8.2.4 dealloc_eq_2()

```
subroutine eq_vars::dealloc_eq_2 (
            class(eq_2_type), intent(inout) eq)
```

Deallocates metric equilibrium quantities.

Parameters

in,out	*eq*	equilibrium to be deallocated

Definition at line 577 of file eq_vars.f90.

Here is the call graph for this function:

B.8.2.5 init_eq_1()

```
subroutine eq_vars::init_eq_1 (
            class(eq_1_type), intent(inout) eq,
            type(grid_type), intent(in) grid,
            logical, intent(in), optional setup_e,
            logical, intent(in), optional setup_f)
```

Initializes new flux equilibrium.

The normal and angular grid can be in any coord. system, as only the grid sizes are used, not the coordinate values.

Optionally, it can be chosen individually whether the E or F(D) quantities are allocated. D means that the derivatives are in F as well. The rationale behind this is that the E quantities are only used in the pre-perturbation phase.

Note

1. The E quantities are not written in eq_ops.print_output_eq().
2. The quantities that do not have a derivative are considered F quantities. Alternatively, all quantities that have only one version, are considered F quantities, such as `rho`, `kappa_n`, ...
3. The maximum derivative degree for flux quantities is one higher than `max_deriv`, because the derivative of some of them appear in the transformation matrices.

Parameters

in,out	*eq*	equilibrium to be initialized

in	*grid*	equilibrium grid
in	*setup↩ _e*	whether to set up E
in	*setup↩ _f*	whether to set up F

Definition at line 173 of file eq_vars.f90.

Here is the call graph for this function:

B.8.2.6 init_eq_2()

```
subroutine eq_vars::init_eq_2 (
                class(eq_2_type), intent(inout) eq,
                type(grid_type), intent(in) grid,
                logical, intent(in), optional setup_e,
                logical, intent(in), optional setup_f)
```

Initializes new metric equilibrium.

See also

For explanation see init_eq_1().

Note

The maximum derivative degree for R, Z and lambda is one higher than `max_deriv`, because their first derivative already appears in g_C and h_C.

Parameters

in,out	*eq*	equilibrium to be initialized
in	*grid*	equilibrium grid
in	*setup↩ _e*	whether to set up E

in	*setup↩ _f*	whether to set up F

Definition at line 272 of file eq_vars.f90.

Here is the call graph for this function:

B.8.3 Variable Documentation

B.8.3.1 b_0

```
real(dp), public eq_vars::b_0
```

derived normalization constant for nondimensionalization

Definition at line 45 of file eq_vars.f90.

B.8.3.2 max_flux_e

```
real(dp), public eq_vars::max_flux_e
```

max. flux in Equilibrium coordinates, set in calc_norm_range_PB3D_in

Definition at line 49 of file eq_vars.f90.

B.8.3.3 max_flux_f

```
real(dp), public eq_vars::max_flux_f
```

max. flux in Flux coordinates, set in calc_norm_range_PB3D_in

Definition at line 50 of file eq_vars.f90.

B.8.3.4 n_alloc_eq_1s

```
integer, public eq_vars::n_alloc_eq_1s
```

nr. of allocated eq_1 variables

Note

Debug version only

Definition at line 53 of file eq_vars.f90.

B.8.3.5 n_alloc_eq_2s

```
integer, public eq_vars::n_alloc_eq_2s
```

nr. of allocated eq_2 variables

Note

Debug version only

Definition at line 55 of file eq_vars.f90.

B.8.3.6 pres_0

```
real(dp), public eq_vars::pres_0
```

independent normalization constant for nondimensionalization

Definition at line 43 of file eq_vars.f90.

B.8.3.7 psi_0

```
real(dp), public eq_vars::psi_0
```

derived normalization constant for nondimensionalization

Definition at line 46 of file eq_vars.f90.

B.8.3.8 r_0

```
real(dp), public eq_vars::r_0
```

independent normalization constant for nondimensionalization

Definition at line 42 of file eq_vars.f90.

B.8.3.9 rho_0

```
real(dp), public eq_vars::rho_0
```

independent normalization constant for nondimensionalization

Definition at line 44 of file eq_vars.f90.

B.8.3.10 t_0

```
real(dp), public eq_vars::t_0
```

derived normalization constant for nondimensionalization

Definition at line 47 of file eq_vars.f90.

B.8.3.11 vac_perm

```
real(dp), public eq_vars::vac_perm = mu_0_original
```

either usual mu_0 (default) or normalized

Definition at line 48 of file eq_vars.f90.

B.9 files_ops Module Reference

Operations related to files !

Functions/Subroutines

- subroutine, public init_files ()
  Initialize the variables for the module.
- integer function, public parse_args ()
  Parses the command line arguments.
- integer function, public open_input ()
  Open the input files.
- integer function, public open_output ()
  Open the output files.
- subroutine, public close_output ()
  Closes the output file.

Variables

- character(len=max_str_ln), dimension(:), allocatable, public opt_args
  optional arguments that can be passed using `--[name]`

B.9.1 Detailed Description

Operations related to files !

B.9.2 Function/Subroutine Documentation

B.9.2.1 close_output()

```
subroutine, public files_ops::close_output
```

Closes the output file.

Definition at line 702 of file files_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.9.2.2 init_files()

```
subroutine, public files_ops::init_files
```

Initialize the variables for the module.

Definition at line 24 of file files_ops.f90.

Here is the caller graph for this function:

B.9.2.3 open_input()

```
integer function, public files_ops::open_input
```

Open the input files.

- input file with user options
- equilibrium file in
  - NetCDF for VMEC
  - plain for HELENA

Returns

ierr

Definition at line 177 of file files_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.9.2.4 open_output()

```
integer function, public files_ops::open_output
```

Open the output files.

- output file `.txt`
- shell commands file `.sh`
- HDF5 file: only for PB3D, not for POST.
- memory usage file `.dat`

Also sets some output variables.

Note

1. memory usage file is only for debug version.
2. There can be resart of a Richardson level for PB3D
3. There can also be a direct jump to the solution for PB3D, if the equilibrium and perturbation phases are already done and saved (see init_files()).
4. In the case of a Richardson restart, PB3D reopens the HDF5 file.

Returns

ierr

Definition at line 528 of file files_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.9.2.5 parse_args()

```
integer function, public files_ops::parse_args
```

Parses the command line arguments.

Note

The input arguments are saved in `command_arg`

Returns

ierr

Definition at line 78 of file files_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.9.3 Variable Documentation

B.9.3.1 opt_args

```
character(len=max_str_ln), dimension(:), allocatable, public files_ops::opt_args
```

optional arguments that can be passed using `--[name]`

Definition at line 19 of file files_ops.f90.

B.10 files_utilities Module Reference

Numerical utilities related to files.

Functions/Subroutines

- integer function, public nextunit (unit)
  Search for available new unit.
- integer function, public skip_comment (file_i, file_name)
  Skips comment when reading a file.
- subroutine, public get_file_info (file_name, file_size, acc_time, mod_time)
  Gets file information.
- character(len=max_str_ln) function, public get_full_pb3d_name (rich_lvl)
  Returns the name of the PB3D output file.
- integer function, public delete_file (file_name)
  Removes a file.
- integer function, public count_lines (file_i)
  Count non-comment lines in a file.

B.10.1 Detailed Description

Numerical utilities related to files.

B.10.2 Function/Subroutine Documentation

B.10.2.1 count_lines()

```
integer function, public files_utilities::count_lines (
            integer, intent(in) file_i)
```

Count non-comment lines in a file.

Parameters

in	*file_i*	file identifier

Definition at line 169 of file files_utilities.f90.

Here is the caller graph for this function:

B.10.2.2 delete_file()

```
integer function, public files_utilities::delete_file (
                character(len=*), intent(inout) file_name)
```

Removes a file.

Returns

istat

Parameters

in,out	*file_name*	the name that is deleted

Definition at line 143 of file files_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.10.2.3 get_file_info()

```
subroutine, public files_utilities::get_file_info (
                character(len=*), intent(in) file_name,
                integer, intent(inout), optional file_size,
                integer, intent(inout), optional acc_time,
                integer, intent(inout), optional mod_time)
```

Gets file information.

The time informations can be converted to strings using the intrinsic function "ctime".

Parameters

in	*file_name*	name of file

in,out	*file_size*	file size
in,out	*acc_time*	file access time
in,out	*mod_time*	file modification time

Definition at line 85 of file files_utilities.f90.

Here is the call graph for this function:

B.10.2.4 get_full_pb3d_name()

```
character(len=max_str_ln) function, public files_utilities::get_full_pb3d_name (
                integer, intent(in), optional rich_lvl)
```

Returns the name of the PB3D output file.

Optionally, the Richardson level can be appended as *R*[lvl].

If not positive, it is ignored.

Definition at line 117 of file files_utilities.f90.

Here is the call graph for this function:

B.10.2.5 nextunit()

```
integer function, public files_utilities::nextunit (
            integer, intent(out), optional unit)
```

Search for available new unit.

`lun_min` and `lun_max` define the range of possible luns to check.

The unit value is returned by the function, and also by the optional argument. This allows the function to be used directly in an open statement, and optionally save the result in a local variable.

If no units are available, -1 is returned.

See also

Adapted from http://fortranwiki.org/fortran/show/newunit

Definition at line 26 of file files_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.10.2.6 skip_comment()

```
integer function, public files_utilities::skip_comment (
                integer, intent(in) file_i,
                character(len=*), intent(in), optional file_name)
```

Skips comment when reading a file.

By comment, a line is meant that starts with the character #.

Returns

ierr

Definition at line 54 of file files_utilities.f90.

Here is the caller graph for this function:

B.11 grid_ops Module Reference

Operations that have to do with the grids and different coordinate systems.

Functions/Subroutines

- integer function, public calc_norm_range (style, in_limits, eq_limits, x_limits, sol_limits, r_f_eq, r_f_x, r_f_sol, jq)

  Calculates normal range for the input grid, the equilibrium grid and/or the solution grid.
- integer function, public setup_grid_eq (grid_eq, eq_limits)

  Sets up the equilibrium grid.
- integer function, public setup_grid_eq_b (grid_eq, grid_eq_b, eq, only_half_grid)

  Sets up the field-aligned equilibrium grid.
- integer function, public setup_grid_x (grid_eq, grid_x, r_f_x, x_limits)

  Sets up the general perturbation grid, in which the perturbation variables are calculated.
- integer function, public setup_grid_sol (grid_eq, grid_x, grid_sol, r_f_sol, sol_limits)

  Sets up the general solution grid, in which the solution variables are calculated.
- integer function, public calc_ang_grid_eq_b (grid_eq, eq, only_half_grid)

  Calculate equilibrium grid that follows magnetic field lines.
- integer function, public redistribute_output_grid (grid, grid_out, no_outer_trim)

  Redistribute a grid to match the normal distribution of solution grid.
- integer function, public magn_grid_plot (grid)

  Plots the grid in real 3-D space.
- integer function, public print_output_grid (grid, grid_name, data_name, rich_lvl, par_div, remove_↩ previous_arrs)

  Print grid variables to an output file.

Variables

- logical, public debug_calc_ang_grid_eq_b = .false.
  plot debug information for calc_ang_grid_eq_b()

B.11.1 Detailed Description

Operations that have to do with the grids and different coordinate systems.

B.11.2 Function/Subroutine Documentation

B.11.2.1 calc_ang_grid_eq_b()

```
integer function, public grid_ops::calc_ang_grid_eq_b (
                type(grid_type), intent(inout) grid_eq,
                type(eq_1_type), intent(in), target eq,
                logical, intent(in), optional only_half_grid)
```

Calculate equilibrium grid that follows magnetic field lines.

This grid is different from the equilibrium grid from setup_grid_eq for HELENA, as the latter is the output grid from HELENA, which is situated in a single poloidal cross-section, as opposed to a really field-aligne grid.

For VMEC, this is not used as the grid is field-aligned from the start.

Note

1. The end-points are included for the grids in the parallel direction. This is to facilitate working with the trapezoidal rule or Simpson's 3/8 rule for integration. This is **NOT** valid in general!
2. by setting the flag `only_half_grid`, only the even points of the parallel grid are calculated, which is useful for higher Richardson levels with VMEC so that only new angular points are calculated and the old ones reused.

Returns

ierr

Parameters

`in,out`	*grid_eq*	equilibrium grid of which to calculate angular part
`in`	*eq*	flux equilibrium variables
`in`	*only_half_grid*	calculate only half grid with even points

Definition at line 797 of file grid_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.11.2.2 calc_norm_range()

```
integer function, public grid_ops::calc_norm_range (
                character(len=*), intent(in) style,
                integer, dimension(2), intent(inout), optional in_limits,
                integer, dimension(2), intent(inout), optional eq_limits,
                integer, dimension(2), intent(inout), optional x_limits,
                integer, dimension(2), intent(inout), optional sol_limits,
                real(dp), dimension(:), intent(inout), optional r_f_eq,
                real(dp), dimension(:), intent(inout), optional, allocatable r_f_x,
                real(dp), dimension(:), intent(inout), optional r_f_sol,
                real(dp), dimension(:), intent(in), optional jq)
```

Calculates normal range for the input grid, the equilibrium grid and/or the solution grid.

General workings, depending on `X_grid_style`:

1 (equilibrium)	2 (solution)	3 (enriched)
calc_eq()	calc_eq()	calc_eq()
	redistribute to sol	add points to optimize
	interpolate to sol	interpolate to X
calc_x()	calc_x()	calc_x()
redistribute to sol	copy to sol	redistribute to sol
interpolate to sol		interpolate to sol

Returns

ierr

Parameters

`in`	*style*	style of calculation (PB3D: in, eq, X or sol; POST)

in,out	*in_limits*	min. and max. index of in grid
in,out	*eq_limits*	min. and max. index of eq grid for this process
in,out	*x_limits*	min. and max. index of X grid for this process
in,out	*sol_limits*	min. and max. index of sol grid for this process
in,out	*r_f_eq*	equilibrium r_F
in,out	*r_f_x*	perturbation r_F
in,out	*r_f_sol*	solution r_F
in	*jq*	q_saf (pol. flux) or rot_t (tor. flux) in total Flux variables

Definition at line 44 of file grid_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.11.2.3 magn_grid_plot()

```
integer function, public grid_ops::magn_grid_plot (
            type(grid_type), intent(in) grid)
```

Plots the grid in real 3-D space.

This creates an animation that can be used by ParaView or VisIt.

The equilibrium grid should contain the fieldline-oriented angles with `ang_1` the parallel angle and `ang_2` the field line label.

See also

See grid_vars.grid_type for a discussion on `ang_1` and `ang_2`.

Note

1. This procedure does not use `n_theta_plot` and `n_zeta_plot` from num_vars, but instead temporarily overwrites them with its own, since it is suposed to be 3-D also in the axisymmetric case.
2. The implementation is currently very slow.

Returns

ierr

Parameters

in	*grid*	fieldline-oriented equilibrium grid

Definition at line 1114 of file grid_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.11.2.4 print_output_grid()

```
integer function, public grid_ops::print_output_grid (
                type(grid_type), intent(in) grid,
                character(len=*), intent(in) grid_name,
                character(len=*), intent(in) data_name,
                integer, intent(in), optional rich_lvl,
                logical, intent(in), optional par_div,
                logical, intent(in), optional remove_previous_arrs)
```

Print grid variables to an output file.

If `rich_lvl` is provided, *R*[`rich_lvl`] is appended to the data name if it is >0.

Optionally, it can be specified that this is a divided parallel grid, corresponding to the variable `eq_jobs_lims` with index `eq_job_nr`. In this case, the total grid size is adjusted to the one specified by `eq_jobs_lims` and the grid is written as a subset.

Note

`grid_` is added in front the data_name.

Returns

ierr

Parameters

in	*grid*	grid variables
in	*grid_name*	name to display
in	*data_name*	name under which to store
in	*rich_lvl*	Richardson level to reconstruct
in	*par_div*	is a parallely divided grid
in	*remove_previous_arrs*	remove previous variables if present

Definition at line 1487 of file grid_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.11.2.5 redistribute_output_grid()

```
integer function, public grid_ops::redistribute_output_grid (
                type(grid_type), intent(in) grid,
                type(grid_type), intent(inout) grid_out,
                logical, intent(in), optional no_outer_trim)
```

Redistribute a grid to match the normal distribution of solution grid.

The routine first calculates the smallest eq range that comprises the sol range. Then, it gets the lowest equilibrium limits able to setup an output grid that starts at index 1. After determining the output grid, it then sends the variables to their new processes using MPI.

Note

1. Only the Flux variables are saved.
2. the redistributed grid has trimmed outer limits, i.e. it starts at 1 and ends at the upper limit of the last process. This can be turned off optionally using `no_outer_trim`.

Returns

ierr

Parameters

in	*grid*	equilibrium grid variables

in,out	*grid_out*	redistributed equilibrium grid variables
in	*no_outer_↩ trim*	do not trim the outer limits

Definition at line 993 of file grid_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.11.2.6 setup_grid_eq()

```
integer function, public grid_ops::setup_grid_eq (
            type(grid_type), intent(inout) grid_eq,
            integer, dimension(2), intent(in) eq_limits)
```

Sets up the equilibrium grid.

The following variables are calculated:

- equilibrium variables (eq)
- perturbation variables (X)

For the implementation of the equilibrium grid the normal part of the grid is always given by the output of the equilibrium code, but the angular part depends on the style:

- VMEC: The output is analytic in the angular coordinates, so field-aligned coordinates are used. Later, in calc_ang_grid_eq_b(), the angular variables are calculated.
- HELENA: The output is given on a poloidal grid (no toroidal dependency due to axisymmetry), which is used.

Returns

ierr

Parameters

in,out	*grid_eq*	equilibrium grid
in	*eq_limits*	min. and max. index of eq grid of this process

Definition at line 528 of file grid_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.11.2.7 setup_grid_eq_b()

```
integer function, public grid_ops::setup_grid_eq_b (
                type(grid_type), intent(inout) grid_eq,
                type(grid_type), intent(inout) grid_eq_b,
                type(eq_1_type), intent(in) eq,
                logical, intent(in), optional only_half_grid)
```

Sets up the field-aligned equilibrium grid.

This serves as a bridge to the solution grid, as it contains the same normal coordinate as the general grid, but the angular coordinates are defined by the solution grid.

Optionally, only half the grid can be calculated (i.e. only the even points), which is used for Richardson levels greater than 1.

Note

In contrast to `setup_grid_eq`, the angular coordinates are also calculated here.

Returns

ierr

Parameters

`in,out`	*grid_eq*	general equilibrium grid

in,out	*grid_eq_b*	field-aligned equilibrium grid
in	*eq*	flux equilibrium variables
in	*only_half_grid*	calculate only half grid with even points

Definition at line 599 of file grid_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.11.2.8 setup_grid_sol()

```
integer function, public grid_ops::setup_grid_sol (
                type(grid_type), intent(in) grid_eq,
                type(grid_type), intent(in) grid_x,
                type(grid_type), intent(inout) grid_sol,
                real(dp), dimension(:), intent(in) r_f_sol,
                integer, dimension(2), intent(in) sol_limits)
```

Sets up the general solution grid, in which the solution variables are calculated.

For the solution grid, only one parallel point is used, but possibly multiple geodesic points, equal to the number of field lines, n_alpha.

Returns

ierr

Parameters

in	*grid_eq*	equilibrium grid

in	*grid_x*	perturbation grid
in,out	*grid_sol*	solution grid
in	*r_f_sol*	points of solution grid
in	*sol_limits*	min. and max. index of sol grid of this process

Definition at line 726 of file grid_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.11.2.9 setup_grid_x()

```
integer function, public grid_ops::setup_grid_x (
                type(grid_type), intent(in) grid_eq,
                type(grid_type), intent(inout) grid_x,
                real(dp), dimension(:), intent(in), allocatable r_f_x,
                integer, dimension(2), intent(in) x_limits)
```

Sets up the general perturbation grid, in which the perturbation variables are calculated.

For `X_grid_style` 1, this grid is identical to the equilibrium grid, and for `X_grid_style` 2,it has the same angular extent but with different normal points, indicated by the variable `r_F_X`.

Returns

ierr

Parameters

in	*grid_eq*	equilibrium grid

in,out	*grid_x*	perturbation grid
in	*r_f_x*	points of perturbation grid
in	*x_limits*	min. and max. index of perturbation grid of this process

Definition at line 654 of file grid_ops.f90.

Here is the caller graph for this function:

B.11.3 Variable Documentation

B.11.3.1 debug_calc_ang_grid_eq_b

```
logical, public grid_ops::debug_calc_ang_grid_eq_b = .false.
```

plot debug information for calc_ang_grid_eq_b()

Note

Debug version only

Definition at line 25 of file grid_ops.f90.

B.12 grid_utilities Module Reference

Numerical utilities related to the grids and different coordinate systems.

Interfaces and Types

- interface coord_f2e

  Converts Flux coordinates $(r,\theta,\zeta)_F$ to Equilibrium coordinates $(r,\theta,\zeta)_E$.
- interface coord_e2f

  Converts Equilibrium coordinates $(r,\theta,\zeta)_E$. to Flux coordinates $(r,\theta,\zeta)_F$.
- interface calc_eqd_grid

  Calculate grid of equidistant points, where optionally the last point can be excluded.
- interface calc_tor_diff

  Calculates the toroidal difference for a magnitude calculated on three toroidal points: two extremities and one in the middle.

Functions/Subroutines

- integer function, public calc_xyz_grid (grid_eq, grid_xyz, x, y, z, l, r)

  Calculates X, Y and Z on a grid `grid_XYZ`, determined through its E(quilibrium) coordinates.
- integer function, public extend_grid_f (grid_in, grid_ext, grid_eq, n_theta_plot, n_zeta_plot, lim_theta↩_plot, lim_zeta_plot)

  Extend a grid angularly.
- integer function, public copy_grid (grid_a, grid_b, lims_b, i_lim)

  Copy a grid A to a new grid B, that was not yet initialized.
- integer function, public calc_int_vol (ang_1, ang_2, norm, j, f, f_int)

  Calculates volume integral on a 3D grid.
- integer function, public trim_grid (grid_in, grid_out, norm_id)

  Trim a grid, removing any overlap between the different regions.
- integer function, public untrim_grid (grid_in, grid_out, size_ghost)

  Untrims a trimmed grid by introducing an asymmetric ghost regions at the right.
- integer function, public calc_vec_comp (grid, grid_eq, eq_1, eq_2, v_com, norm_disc_prec, v_mag, base_name, max_transf, v_flux_tor, v_flux_pol, xyz, compare_tor_pos)

  Calculates contra- and covariant components of a vector in multiple coordinate systems.
- integer function, public calc_n_par_x_rich (n_par_x_rich, only_half_grid)

  Calculates the local number of parallel grid points for this Richardson level, taking into account that it ould be half the actual number.
- integer function, public nufft (x, f, f_f, plot_name)

  calculates the cosine and sine mode numbers of a function defined on a non-regular grid.
- subroutine, public find_compr_range (r_f, lim_r, lim_id)

  finds smallest range that comprises a minimum and maximum value.
- integer function, public calc_arc_angle (grid, eq_1, eq_2, arc, use_e)

  Calculate arclength angle.

Variables

- logical, public debug_calc_int_vol = .false.

  plot debug information for calc_int_vol()
- logical, public debug_calc_vec_comp = .false.

  plot debug information for calc_vec_comp()

B.12.1 Detailed Description

Numerical utilities related to the grids and different coordinate systems.

B.12.2 Function/Subroutine Documentation

B.12.2.1 calc_arc_angle()

```
integer function, public grid_utilities::calc_arc_angle (
                type(grid_type), intent(in) grid,
                type(eq_1_type), intent(in) eq_1,
                type(eq_2_type), intent(in) eq_2,
                real(dp), dimension(:,:,:), intent(inout), allocatable arc,
                logical, intent(in), optional use_e)
```

Calculate arclength angle.

This angle is based on the calculation of the arclength from the start of the grid, and then normalizing it from $\min(\gamma) \ldots \max(\gamma)$, where γ is the parallel angle.

By default, the Flux variables are used, but this can be changed with `use_E`.

Note

For field-aligned grids the projection is taken on the poloidal cross-section. For non-axisymmetric equilibria, this makes little sense.

Returns

ierr

Parameters

in	*grid*	equilibrium grid
in	*eq_1*	flux equilibrium variables
in	*eq_2*	metric equilibrium variables
in,out	*arc*	arclength angle
in	*use_e*	use E variables instead of F

Definition at line 3038 of file grid_utilities.f90.

Here is the call graph for this function:

B.12.2.2 calc_int_vol()

```
integer function, public grid_utilities::calc_int_vol (
                real(dp), dimension(:,:,:), intent(in) ang_1,
                real(dp), dimension(:,:,:), intent(in) ang_2,
                real(dp), dimension(:), intent(in) norm,
                real(dp), dimension(:,:,:), intent(in) j,
                complex(dp), dimension(:,:,:,:), intent(in) f,
                complex(dp), dimension(:), intent(inout) f_int)
```

Calculates volume integral on a 3D grid.

Two angular and one normal variable have to be provided on a the grid.

If the `i`'th dimension of the grid is equal to one, the function to be integrated is assumed not to vary in this dimension.

Furthermore, if `i` is 1 or 2, the corresponding `i`'th (angular) variable is the only variable that is assumed to vary in that dimension. The other angular variable as well as the normal variable are assumed to be constant like the function itself, resulting in a factor 2π. However, if `i` is 3, an error is displayed as this does not represent a physical situation.

A common case through which to understand this is the axisymmetric case where the first angular variable θ varies in the dimensions 1 and 3, the second angular variable ζ varies only in the dimension 2, and the normal variable only varies in the dimension 3.

Alternatively, there is the case of a grid-aligned set of coordinates θ and α, where the first dimension corresponds to the direction along the magnetic field line, the second to the geodesical direction and the third to the normal direction. If the calculations for different field lines are decoupled, the variation in the second dimension is not taken into account and no integration happens along it.

Internally, the angular variables and the normal variable are related to the coordinates (x, y, z) that correspond to the three dimensions. They thus form a computational orthogonal grid to which the original coordinates are related through the transformation of Jacobians:

$$\mathcal{J}_{xyz} = \mathcal{J}_{\mathsf{F}} \frac{\partial r_{\mathsf{F}}}{\partial z} \left(\frac{\partial \gamma_1}{\partial x} \frac{\gamma_2}{\partial y} - \frac{\partial \gamma_1}{\partial y} \frac{\gamma_2}{\partial x} \right),$$

where γ_1 and γ_2 are `ang_1` and `ang_2`, so that the integral becomes

$$\sum_{x,y,z} \left[f(x, y, z)\, \mathcal{J}_{\mathsf{F}} \frac{\partial r_{\mathsf{F}}}{\partial z} \left(\frac{\partial \gamma_1}{\partial x} \frac{\gamma_2}{\partial y} - \frac{\partial \gamma_1}{\partial y} \frac{\gamma_2}{\partial x} \right) \Delta_x \Delta_y \Delta_z \right]$$

where Δ_x, Δ_y and Δ_z are all trivially equal to 1.

The integrand has to be evaluated at the intermediate positions inside the cells. This is done by taking the average of the $2^3 = 8$ points for `fj` = $f J_{\mathsf{F}}$ as well as the transformation of the Jacobian to (x, y, z) coordinates.

See also

See grid_vars.grid_type for a discussion on `ang_1` and `ang_2`.

Note

1. if the coordinates are independent, this method is equivalent to the repeated numerical integration using the trapezoidal method, **NOT** Simpson's 3/8 rule!
2. by setting `debug_calc_int_vol`, this method can be compared to the trapezoidal and simple method for independent coordinates, again **NOT** for Simpson's 3/8 rule!
3. The Simpson's 3/8 rule could be developed but it is not of great importance.

Returns

ierr

Parameters

`in`	*ang↩ _1*	coordinate variable γ_1

in	*ang↩ _2*	coordinate variable γ_2
in	*norm*	coordinate variable r_F
in	*j*	Jacobian
in	*f*	input f(n_par,n_geo,n_r,size_X^2)
in,out	*f_int*	output integrated f

Definition at line 1319 of file grid_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.12.2.3 calc_n_par_x_rich()

```
integer function, public grid_utilities::calc_n_par_x_rich (
            integer, intent(inout) n_par_x_rich,
            logical, intent(in), optional only_half_grid)
```

Calculates the local number of parallel grid points for this Richardson level, taking into account that it ould be half the actual number.

See also

calc_ang_grid_eq_b().

Returns

ierr

Parameters

in,out	*n_par_x_rich*	n_par_X for this Richardson level

in	*only_half_grid*	calculate only half grid with even points

Definition at line 2861 of file grid_utilities.f90.

Here is the caller graph for this function:

B.12.2.4 calc_vec_comp()

```
integer function, public grid_utilities::calc_vec_comp (
                type(grid_type), intent(in) grid,
                type(grid_type), intent(in) grid_eq,
                type(eq_1_type), intent(in) eq_1,
                type(eq_2_type), intent(in) eq_2,
                real(dp), dimension(:,:,:,:,:), intent(inout) v_com,
                integer, intent(in) norm_disc_prec,
                real(dp), dimension(:,:,:), intent(inout), optional v_mag,
                character(len=*), intent(in), optional base_name,
                integer, intent(in), optional max_transf,
                real(dp), dimension(:,:), intent(inout), optional, allocatable v_flux_tor,
                real(dp), dimension(:,:), intent(inout), optional, allocatable v_flux_pol,
                real(dp), dimension(:,:,:,:), intent(in), optional xyz,
                logical, intent(in), optional compare_tor_pos)
```

Calculates contra- and covariant components of a vector in multiple coordinate systems.

Chain of coordinate systems considered:

1. Flux F: (α, ψ, θ),
2. Magnetic M: (ψ, θ, ζ),
   - H: (ψ, θ, ϕ) if HELENA is used,
   - V: (r, θ, ζ) if VMEC is used,
3. Cylindrical C: (R, φ, Z),
4. Cartesian X: (X, Y, Z), as well as the magnitude, starting from the input in Flux coordinates. Note that the Cartesian components in X, Y and Z can be used to plot the real vector and that covariant components should be equal to contravariant components in this coordinate system.

Also, the fluxes can be calculated and plot by providing a `base_name`.

By default, the cartesian components are returned, but this can be indicated differently by providing `max_transf`.

Note

1. Plots for different Richardson levels can be combined to show the total grid by just plotting them all individually and sequentially.
2. The metric factors and transformation matrices have to be allocated. They can be calculated using the routines from eq_ops, for `deriv = [0,0,0]`.
3. For VMEC, the trigonometric factors of `grid_XYZ` must be calculated beforehand. Optionally, by providing `X`, `Y` and `Z`, the ones calculated in this routine are overwritten. This is useful for, for example, slab geometries.
4. The normalization factors are taken into account and the output is transformed back to unnormalized values.

Returns

ierr

Parameters

in	*grid*	grid on which vector is calculated
in	*grid_eq*	grid on which equilibrium variables are calculated
in	*eq_1*	flux equilibrium quantities
in	*eq_2*	metric equilibrium variables
in,out	*v_com*	covariant and contravariant components `(dim1,dim2,dim3,3,2)`
in	*norm_disc_prec*	precision for normal derivatives
in,out	*v_mag*	magnitude `(dim1,dim2,dim3)`
in	*base_name*	base name for output plotting
in	*max_transf*	maximum transformation level (2: Magnetic, 3: Equilibrium, 4: Cylindrical, 5: Cartesian [def])
in,out	*v_flux_pol*	poloidal flux as function of normal coordinate for all toroidal positions
in,out	*v_flux_tor*	toroidal flux as function of normal coordinate for all poloidal positions
in	*xyz*	X, Y and Z of grid
in	*compare_tor_pos*	compare toroidal positions

Definition at line 1856 of file grid_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.12.2.5 calc_xyz_grid()

```
integer function, public grid_utilities::calc_xyz_grid (
                type(grid_type), intent(in) grid_eq,
                type(grid_type), intent(in) grid_xyz,
                real(dp), dimension(:,:,:), intent(inout) x,
                real(dp), dimension(:,:,:), intent(inout) y,
                real(dp), dimension(:,:,:), intent(inout) z,
                real(dp), dimension(:,:,:), intent(inout), optional l,
                real(dp), dimension(:,:,:), intent(inout), optional r)
```

Calculates X, Y and Z on a grid `grid_XYZ`, determined through its E(quilibrium) coordinates.

Furthermore, a grid `grid_eq` must be provided, which is the grid in which the variables concerning R and Z are tabulated, i.e. the full equilibrium grid in E(quilibrium) coordinates.

Of this grid, however, only `r_E` is used, and the rest ignored. It can therefore be provided without the angular part, i.e. by reconstructing it with a subset.

If VMEC is the equilibrium model, this routine also optionally calculates λ on the grid, as this is also needed some times. for HELENA this variable is not used.

Optionally, R and λ (only for VMEC) can be returned.

Note

1. For VMEC, the trigonometric factors of `grid_XYZ` must be calculated beforehand.
2. The normalization factor `R_0` for length is taken into account and the output is transformed back to unnormalized values.

Returns

ierr

Parameters

in	*grid_eq*	equilibrium grid
in	*grid_xyz*	grid for which to calculate X, Y, Z and optionally L
in,out	*x*	X of grid
in,out	*y*	Y of grid
in,out	*z*	Z of grid
in,out	*l*	λ of grid
in,out	*r*	R of grid

Definition at line 798 of file grid_utilities.f90.

Here is the caller graph for this function:

B.12.2.6 copy_grid()

```
integer function, public grid_utilities::copy_grid (
            class(grid_type), intent(in) grid_a,
            class(grid_type), intent(inout) grid_b,
            integer, dimension(3,2), intent(in), optional lims_b,
            integer, dimension(2), intent(in), optional i_lim)
```

Copy a grid A to a new grid B, that was not yet initialized.

This new grid can contain a subsection of the previous grid in all dimensions. It can also be a divided grid, by providing the limits

Note

1. The normal limits for the divided grid should be given in terms of the normal dimension of `grid` B.
2. if the grids are not purely normal, the procedure can currently only handle the copying of a `grid_A` that is not divided.

Returns

ierr

Parameters

in	*grid↩ _a*	grid to be initialized
in,out	*grid↩ _b*	grid to be initialized
in	*lims↩ _b*	ranges for subset of grid
in	*i_lim*	min. and max. local normal index

Definition at line 1188 of file grid_utilities.f90.

Here is the caller graph for this function:

B.12.2.7 extend_grid_f()

```
integer function, public grid_utilities::extend_grid_f (
                type(grid_type), intent(in) grid_in,
                type(grid_type), intent(inout) grid_ext,
                type(grid_type), intent(in), optional grid_eq,
                integer, intent(in), optional n_theta_plot,
                integer, intent(in), optional n_zeta_plot,
                real(dp), dimension(2), intent(in), optional lim_theta_plot,
                real(dp), dimension(2), intent(in), optional lim_zeta_plot)
```

Extend a grid angularly.

This is done using equidistant variables of `n_theta_plot` and `n_zeta_plot` angular and own `loc_n_r` points in F coordinates.

Optionally, the grid can also be converted to E coordinates if the equilibrium grid is provided. This is required for many operations, such as the calculation of X, Y and Z through calc_XYZ_grid().

Note

For VMEC, it can be slow, as grid_utilities.coord_f2e() is used.

Returns

ierr

Parameters

in	*grid_in*	grid to be extended
in,out	*grid_ext*	extended grid
in	*grid_eq*	equilibrium grid
in	*n_theta_plot*	number of poins in theta direction
in	*n_zeta_plot*	number of poins in zeta direction
in	*lim_theta_plot*	limits in theta
in	*lim_zeta_plot*	limits in zeta

Definition at line 1094 of file grid_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.12.2.8 find_compr_range()

```
subroutine, public grid_utilities::find_compr_range (
            real(dp), dimension(:), intent(in) r_f,
            real(dp), dimension(2), intent(in) lim_r,
            integer, dimension(2), intent(inout) lim_id)
```

finds smallest range that comprises a minimum and maximum value.

Parameters

in	*r_f*	all values of coordinate
in	*lim_r*	limiting range
in,out	*lim_id*	limiting indices

Definition at line 2994 of file grid_utilities.f90.

Here is the caller graph for this function:

B.12.2.9 nufft()

```
integer function, public grid_utilities::nufft (
            real(dp), dimension(:), intent(in) x,
            real(dp), dimension(:), intent(in) f,
            real(dp), dimension(:,:), intent(inout), allocatable f_f,
            character(len=*), intent(in), optional plot_name)
```

calculates the cosine and sine mode numbers of a function defined on a non-regular grid.

If a plot name is provided, the modes are plotted.

Note

The fundamental interval is assumed to be $0 \ldots 2\pi$ but there should be no overlap between the first and last point.

Returns

ierr

Parameters

in	*x*	coordinate values

in	*f*	function values
in,out	*f_f*	Fourier modes for cos and sin
in	*plot_name*	name of possible plot

Definition at line 2900 of file grid_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.12.2.10 trim_grid()

```
integer function, public grid_utilities::trim_grid (
            type(grid_type), intent(in) grid_in,
            type(grid_type), intent(inout) grid_out,
            integer, dimension(2), intent(inout), optional norm_id)
```

Trim a grid, removing any overlap between the different regions.

by default, the routine assumes a symmetric ghost region and cuts as many grid points from the end of the previous process as from the beginning of the next process, but if the number of overlapping grid points is odd, the next process looses one more point.

optionally, the trimmed indices in the normal direction can be provided in `norm_id`, i.e. the indices in the old, untrimmed grid that correspond to the start and end indices of the trimmed grid. E.g. if

- proc 0: 3 ... 25
- proc 1: 20 ... 50

then the trimmed grid will be:

- proc 0: 3 ... 22
- proc 1: 23 ... 50

which is shifted down by 2 to

- proc 0: 1 ... 20
- proc 1: 21 ... 48

in the trimmed grid. The indices of the previous step (3 & 22 and 23 & 50) are saved in `norm_id`.

Returns

ierr

Parameters

`in`	*grid_in*	input grid
`in,out`	*grid_out*	trimmed grid
`in,out`	*norm_id*	normal indices corresponding to trimmed part

Definition at line 1635 of file grid_utilities.f90.

Here is the caller graph for this function:

B.12.2.11 untrim_grid()

```
integer function, public grid_utilities::untrim_grid (
            type(grid_type), intent(in) grid_in,
            type(grid_type), intent(inout) grid_out,
            integer, intent(in) size_ghost)
```

Untrims a trimmed grid by introducing an asymmetric ghost regions at the right.

The width of the ghost region has to be provided.

Note

1. the ghosted grid should be deallocated (with dealloc_grid()).
2. the input grid has to be trimmed.

Returns

ierr

Parameters

in	*grid_in*	input grid
in,out	*grid_out*	ghosted grid
in	*size_↩ ghost*	width of ghost region

Definition at line 1763 of file grid_utilities.f90.

Here is the caller graph for this function:

B.12.3 Variable Documentation

B.12.3.1 debug_calc_int_vol

```
logical, public grid_utilities::debug_calc_int_vol = .false.
```

plot debug information for calc_int_vol()

Note

Debug version only

Definition at line 25 of file grid_utilities.f90.

B.12.3.2 debug_calc_vec_comp

```
logical, public grid_utilities::debug_calc_vec_comp = .false.
```

plot debug information for calc_vec_comp()

Note

Debug version only

Definition at line 27 of file grid_utilities.f90.

B.13 grid_vars Module Reference

Variables pertaining to the different grids used.

Interfaces and Types

- type grid_type
  Type for grids.

Functions/Subroutines

- integer function init_grid (grid, n, i_lim, divided)
  Initializes a new grid.
- subroutine dealloc_grid (grid)
  Deallocates a grid.
- integer function copy_grid (grid_i, grid_o)
  Deep copy of a grid.

Variables

- integer, public n_r_in
  nr. of normal points in input grid
- integer, public n_r_eq
  nr. of normal points in equilibrium grid
- integer, public n_r_x
  nr. of normal points in perturbation grid
- integer, public n_r_sol
  nr. of normal points in solution grid
- integer, public n_alpha
  nr. of field-lines
- real(dp), public min_par_x
  min. of parallel coordinate [π] in field-aligned grid
- real(dp), public max_par_x
  max. of parallel coordinate [π] in field-aligned grid
- real(dp), public min_alpha
  min. of field-line label [π] in field-aligned grid
- real(dp), public max_alpha
  max. of field-line label [π] in field-aligned grid
- real(dp), dimension(:), allocatable, public alpha
  field line label alpha
- integer, public n_alloc_grids
  nr. of allocated grids
- integer, public n_alloc_discs
  nr. of allocated discretizations

B.13.1 Detailed Description

Variables pertaining to the different grids used.

B.13.2 Function/Subroutine Documentation

B.13.2.1 copy_grid()

```
integer function grid_vars::copy_grid (
                class(grid_type), intent(in) grid_i,
                type(grid_type), intent(inout) grid_o)
```

Deep copy of a grid.

Note

Does not copy possible trigoniometric factors.

Returns

ierr

Parameters

in	*grid↩ _i*	grid to be copied
in,out	*grid↩ _o*	copied grid

Definition at line 266 of file grid_vars.f90.

Here is the call graph for this function:

B.13.2.2 dealloc_grid()

```
subroutine grid_vars::dealloc_grid (
            class(grid_type), intent(inout) grid)
```

Deallocates a grid.

Parameters

in,out	*grid*	grid to be deallocated

Definition at line 202 of file grid_vars.f90.

Here is the call graph for this function:

B.13.2.3 init_grid()

```
integer function grid_vars::init_grid (
            class(grid_type), intent(inout) grid,
            integer, dimension(3), intent(in) n,
            integer, dimension(2), intent(in), optional i_lim,
            logical, intent(in), optional divided)
```

Initializes a new grid.

Optionally, the local limits can be provided for a divided grid.

Optionally, it can be set whether the grid is divided or not. A situation where this is useful is when only a subset of MPI processes is used to calculate the solution in SLEPC. In this case, the extra ranks all only contain the last grid point, and the last used process has as upper limit this same grid point. This way, all the procedures are reusable, but in this case if only one process is used, this procedure becomes confused and sets this process to undivided. In most cases, this functionality probably does not need to be used.

Returns

ierr

Parameters

in,out	*grid*	grid to be initialized

in	*n*	tot. nr. of points (par,r,alpha)
in	*i_lim*	min. and max. local normal index
in	*divided*	divided grid or not

Definition at line 101 of file grid_vars.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.13.3 Variable Documentation

B.13.3.1 alpha

```
real(dp), dimension(:), allocatable, public grid_vars::alpha
```

field line label alpha

Definition at line 28 of file grid_vars.f90.

B.13.3.2 max_alpha

```
real(dp), public grid_vars::max_alpha
```

max. of field-line label [π] in field-aligned grid

Definition at line 27 of file grid_vars.f90.

B.13.3.3 max_par_x

```
real(dp), public grid_vars::max_par_x
```

max. of parallel coordinate [π] in field-aligned grid

Definition at line 25 of file grid_vars.f90.

B.13.3.4 min_alpha

```
real(dp), public grid_vars::min_alpha
```

min. of field-line label [π] in field-aligned grid

Definition at line 26 of file grid_vars.f90.

B.13.3.5 min_par_x

```
real(dp), public grid_vars::min_par_x
```

min. of parallel coordinate [π] in field-aligned grid

Definition at line 24 of file grid_vars.f90.

B.13.3.6 n_alloc_discs

```
integer, public grid_vars::n_alloc_discs
```

nr. of allocated discretizations

Note

Debug version only

Definition at line 31 of file grid_vars.f90.

B.13.3.7 n_alloc_grids

```
integer, public grid_vars::n_alloc_grids
```

nr. of allocated grids

Note

Debug version only

Definition at line 30 of file grid_vars.f90.

B.13.3.8 n_alpha

```
integer, public grid_vars::n_alpha
```

nr. of field-lines

Definition at line 23 of file grid_vars.f90.

B.13.3.9 n_r_eq

```
integer, public grid_vars::n_r_eq
```

nr. of normal points in equilibrium grid

Definition at line 20 of file grid_vars.f90.

B.13.3.10 n_r_in

```
integer, public grid_vars::n_r_in
```

nr. of normal points in input grid

Definition at line 19 of file grid_vars.f90.

B.13.3.11 n_r_sol

```
integer, public grid_vars::n_r_sol
```

nr. of normal points in solution grid

Definition at line 22 of file grid_vars.f90.

B.13.3.12 n_r_x

```
integer, public grid_vars::n_r_x
```

nr. of normal points in perturbation grid

Definition at line 21 of file grid_vars.f90.

B.14 hdf5_ops Module Reference

Operations on HDF5 and XDMF variables.

Interfaces and Types

- interface reset_hdf5_item

  Resets an HDF5 XDMF item.

Functions/Subroutines

- integer function, public open_hdf5_file (file_info, file_name, sym_type, descr, ind_plot, cont_plot)

  Opens an HDF5 file and accompanying xmf file and returns the handles.
- integer function, public close_hdf5_file (file_info, ind_plot, cont_plot)

  Closes an HDF5 file and writes the accompanying xmf file.
- integer function, public add_hdf5_item (file_info, xdmf_item, reset, ind_plot)

  Add an XDMF item to a HDF5 file.
- integer function, public print_hdf5_3d_data_item (dataitem_id, file_info, var_name, var, dim_tot, loc↩_dim, loc_offset, init_val, ind_plot, cont_plot)

  Prints an HDF5 data item.
- subroutine, public merge_hdf5_3d_data_items (merged_id, dataitem_ids, var_name, dim_tot, reset, ind_plot, cont_plot)

  Joins dataitems in a vector.
- subroutine, public print_hdf5_att (att_id, att_dataitem, att_name, att_center, att_type, reset, ind_plot)

  Prints an HDF5 attribute.
- subroutine, public print_hdf5_top (top_id, top_type, top_n_elem, ind_plot)

  Prints an HDF5 topology.
- subroutine, public print_hdf5_geom (geom_id, geom_type, geom_dataitems, reset, ind_plot)

  Prints an HDF5 geometry.
- integer function, public print_hdf5_grid (grid_id, grid_name, grid_type, grid_time, grid_top, grid_geom, grid_atts, grid_grids, reset, ind_plot)

  Prints an HDF5 grid.
- integer function, public create_output_hdf5 (hdf5_name)

  Creates an HDF5 output file.
- integer function, public print_hdf5_arrs (vars, pb3d_name, head_name, rich_lvl, disp_info, ind_print, remove_previous_arrs)

  Prints a series of arrays, in the form of an array of pointers, to an HDF5 file.
- integer function, public read_hdf5_arr (var, pb3d_name, head_name, var_name, rich_lvl, disp_info, lim_loc)

  Reads a PB3D output file in HDF5 format.

Variables

- logical, public debug_hdf5_ops = .false.

  set to true to debug general information
- logical, public debug_print_hdf5_arrs = .false.

  set to true to debug print_HDF5_arrs

B.14.1 Detailed Description

Operations on HDF5 and XDMF variables.

Note

Notes about XDMF:

- Collections can be spatial or temporal. If a variable is to be evolved in time or if its domain is decomposed (e.g. the same physical variable is defined in multiple HDF5 variables), then the attribute name of this variable has to be the same for all the grids in the collection.
- If no collection is used, but just multiple grids, the attribute can be different but does not have to be, as the different grid are distinguished by their different grid names, not by the attribute of the variables they contain.
- The selection of hyperslabs is used, as described here: http://davis.lbl.gov/Manuals/HDF5-1.8.7/UG/12_Dataspaces.html
- Chunking is used for partial I/O, as described here: https://www.hdfgroup.org/HDF5/doc/Advanced/Chunking/ As no chunk cache is reused, the w0 setting should be 1. Furthermore, the number of slots is chosen to be equal to the number of chunks. And Finally, the chunk size is chosen to be the size of the previous dimensions, if it does not exceed 4GB
- Individual writes or plots are done now using the standard I/O driver, as using MPI does not allow for more than 2GB. This is important for the HELENA version of PB3D where X_2 variables get larger than that. https://www.hdfgroup.org/HDF5/doc/UG/OldHtmlSource/UG_frame08TheFile.html
- Have a look in Documentation/XDMF_HDF5.pdf for an example.

B.14.2 Function/Subroutine Documentation

B.14.2.1 add_hdf5_item()

```
integer function, public hdf5_ops::add_hdf5_item (
                type(hdf5_file_type), intent(inout) file_info,
                type(xml_str_type), intent(inout) xdmf_item,
                logical, intent(in), optional reset,
                logical, intent(in), optional ind_plot)
```

Add an XDMF item to a HDF5 file.

Note

This should only be used with grids (or for topologies and geometries that are used throughout)

Returns

ierr

Parameters

in,out	*file_info*	info about HDF5 file

in,out	*xdmf_↩ item*	XDMF item to add
in	*reset*	if .true., XDMF_item is reset
in	*ind_plot*	whether individual plot

Definition at line 338 of file HDF5_ops.f90.

Here is the caller graph for this function:

B.14.2.2 close_hdf5_file()

```
integer function, public hdf5_ops::close_hdf5_file (
            type(hdf5_file_type), intent(inout) file_info,
            logical, intent(in), optional ind_plot,
            logical, intent(in), optional cont_plot)
```

Closes an HDF5 file and writes the accompanying xmf file.

Returns

ierr

Parameters

in,out	*file_info*	info about HDF5 file
in	*ind_plot*	whether individual plot
in	*cont_↩ plot*	continued plot

Definition at line 263 of file HDF5_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.14.2.3 create_output_hdf5()

```
integer function, public hdf5_ops::create_output_hdf5 (
                character(len=*), intent(in) hdf5_name)
```

Creates an HDF5 output file.

Parameters

in	*hdf5_name*	name of HDF5 file

Definition at line 1074 of file HDF5_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.14.2.4 merge_hdf5_3d_data_items()

```
subroutine, public hdf5_ops::merge_hdf5_3d_data_items (
                type(xml_str_type), intent(inout) merged_id,
                type(xml_str_type), dimension(:), intent(inout) dataitem_ids,
                character(len=*), intent(in) var_name,
                integer, dimension(3), intent(in) dim_tot,
                logical, intent(in), optional reset,
                logical, intent(in), optional ind_plot,
                logical, intent(in), optional cont_plot)
```

Joins dataitems in a vector.

Parameters

in,out	*merged_id*	ID of merged data item
in,out	*dataitem_ids*	ID of data items
in	*var_name*	name of variable
in	*dim_tot*	total dimensions of variable
in	*reset*	whether to reset dataitems
in	*ind_plot*	whether individual plot
in	*cont_plot*	continued plot

Definition at line 601 of file HDF5_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.14.2.5 open_hdf5_file()

```
integer function, public hdf5_ops::open_hdf5_file (
                type(hdf5_file_type), intent(inout) file_info,
                character(len=*), intent(in) file_name,
                integer, intent(inout), optional sym_type,
                character(len=*), intent(in), optional descr,
                logical, intent(in), optional ind_plot,
                logical, intent(in), optional cont_plot)
```

Opens an HDF5 file and accompanying xmf file and returns the handles.

Optionally, a description of the file can be provided.

Also, the plot can be done for only one process, setting the variable `ind_plot`.

Furthermore, if the plot is a continuation, using `cont_plot`, the previous plot is opened and `sym_type` is returned.

Returns

ierr

Parameters

`in,out`	*file_info*	info about HDF5 file
`in`	*file_name*	name of HDF5 file
`in,out`	*sym_type*	symmetry type
`in`	*descr*	description of file
`in`	*ind_plot*	whether individual plot
`in`	*cont_plot*	continued plot

Definition at line 76 of file HDF5_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.14.2.6 print_hdf5_3d_data_item()

```
integer function, public hdf5_ops::print_hdf5_3d_data_item (
                type(xml_str_type), intent(inout) dataitem_id,
                type(hdf5_file_type), intent(in) file_info,
                character(len=*), intent(in) var_name,
                real(dp), dimension(:,:,:), intent(in) var,
                integer, dimension(3), intent(in) dim_tot,
                integer, dimension(3), intent(in), optional loc_dim,
                integer, dimension(3), intent(in), optional loc_offset,
                real(dp), intent(in), optional init_val,
                logical, intent(in), optional ind_plot,
                logical, intent(in), optional cont_plot)
```

Prints an HDF5 data item.

If this is a parallel data item, the group dimension and offset have to be specified as well.

Returns

ierr

Parameters

in,out	*dataitem_id*	ID of data item
in	*file_info*	info about HDF5 file
in	*var_name*	name of variable
in	*var*	variable to write
in	*dim_tot*	total dimensions of variable
in	*loc_dim*	dimensions in this group
in	*loc_offset*	offset in this group
in	*init_val*	initial fill value
in	*ind_plot*	whether individual plot
in	*cont_plot*	continued plot

Definition at line 393 of file HDF5_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.14.2.7 print_hdf5_arrs()

```
integer function, public hdf5_ops::print_hdf5_arrs (
                type(var_1d_type), dimension(:), intent(in) vars,
                character(len=*), intent(in) pb3d_name,
                character(len=*), intent(in) head_name,
                integer, intent(in), optional rich_lvl,
                logical, intent(in), optional disp_info,
                logical, intent(in), optional ind_print,
                logical, intent(in), optional remove_previous_arrs)
```

Prints a series of arrays, in the form of an array of pointers, to an HDF5 file.

Optionally, output can be given about the variable being written. Also, if `rich_lvl` is provided, *R*[`rich_lvl`] is appended to the head name if it is >0.

Also optionally, previously encountered arrays can be removed. This is to be used with care, as it may disturb the internal workings of PB3D. Currently (v2.15) it is used only for the specific case of jumping to solutions for `X_grid_style` 1 or 3, when writing solutions and solution grids.

Note

See https://www.hdfgroup.org/HDF5/doc/UG/12_Dataspaces.html, 7.4.2.3 for an explanation of the selection of the dataspaces.

Returns

ierr

Parameters

in	*vars*	variables to write

in	*pb3d_name*	name of PB3D file
in	*head_name*	head name of variables
in	*rich_lvl*	Richardson level to append to file name
in	*disp_info*	display additional information about variable being read
in	*ind_print*	individual write (possibly partial I/O)
in	*remove_previous_arrs*	remove previous variables if present

Definition at line 1130 of file HDF5_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.14.2.8 print_hdf5_att()

```
subroutine, public hdf5_ops::print_hdf5_att (
            type(xml_str_type), intent(inout) att_id,
```

```
type(xml_str_type), intent(inout) att_dataitem,
character(len=*), intent(in) att_name,
integer, intent(in) att_center,
integer, intent(in) att_type,
logical, intent(in), optional reset,
logical, intent(in), optional ind_plot)
```

Prints an HDF5 attribute.

Parameters

in,out	*att_id*	ID of attribute
in,out	*att_dataitem*	dataitem of attribute
in	*att_name*	name of attribute
in	*att_center*	center of attribute
in	*att_type*	type of attribute
in	*reset*	whether to reset dataitems
in	*ind_plot*	whether individual plot

Definition at line 694 of file HDF5_ops.f90.

Here is the caller graph for this function:

B.14.2.9 print_hdf5_geom()

```
subroutine, public hdf5_ops::print_hdf5_geom (
                type(xml_str_type), intent(inout) geom_id,
                integer, intent(in) geom_type,
                type(xml_str_type), dimension(:), intent(inout) geom_dataitems,
                logical, intent(in), optional reset,
                logical, intent(in), optional ind_plot)
```

Prints an HDF5 geometry.

Parameters

in,out	*geom_id*	ID of geometry

in	*geom_type*	type of geometry
in,out	*geom_dataitems*	data items of geometry
in	*reset*	whether to reset dataitems
in	*ind_plot*	whether individual plot

Definition at line 804 of file HDF5_ops.f90.

Here is the caller graph for this function:

B.14.2.10 print_hdf5_grid()

```
integer function, public hdf5_ops::print_hdf5_grid (
                type(xml_str_type), intent(inout) grid_id,
                character(len=*), intent(in) grid_name,
                integer, intent(in) grid_type,
                real(dp), intent(in), optional grid_time,
                type(xml_str_type), optional grid_top,
                type(xml_str_type), optional grid_geom,
                type(xml_str_type), dimension(:), optional grid_atts,
                type(xml_str_type), dimension(:), optional grid_grids,
                logical, intent(in), optional reset,
                logical, intent(in), optional ind_plot)
```

Prints an HDF5 grid.

If this is is a uniform grid, the geometry and topology have to be specified, or else it will be assumed that it is already present in the XDMF domain, and reused.

If the grid is a collection grid, the grids in the collection have to be specified.

Optionally, also a time can be specified (for the grids in a collection grid).

Returns

ierr

Parameters

in,out	*grid_id*	ID of grid

in	*grid_name*	name
in	*grid_type*	type
in	*grid_time*	time of grid
	grid_top	topology
	grid_geom	geometry
	grid_atts	attributes
	grid_grids	grids
in	*reset*	whether top, geom and atts and grids are reset
in	*ind_plot*	whether individual plot

Definition at line 884 of file HDF5_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.14.2.11 print_hdf5_top()

```
subroutine, public hdf5_ops::print_hdf5_top (
                type(xml_str_type), intent(inout) top_id,
```

```
integer, intent(in) top_type,
integer, dimension(:), intent(in) top_n_elem,
logical, intent(in), optional ind_plot)
```

Prints an HDF5 topology.

Note

Currently only structured grids supported.

Parameters

in,out	*top_id*	ID of topology
in	*top_type*	type
in	*top_n_elem*	nr. of elements
in	*ind_plot*	whether individual plot

Definition at line 754 of file HDF5_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.14.2.12 read_hdf5_arr()

```
integer function, public hdf5_ops::read_hdf5_arr (
                type(var_1d_type), intent(inout) var,
                character(len=*), intent(in) pb3d_name,
                character(len=*), intent(in) head_name,
                character(len=*), intent(in) var_name,
                integer, intent(in), optional rich_lvl,
                logical, intent(in), optional disp_info,
                integer, dimension(:,:), intent(in), optional lim_loc)
```

Reads a PB3D output file in HDF5 format.

This happens in a non-parallel way. By default, all variables are read, but an array of strings with acceptable variable names can be passed.

Optionally, output can be given about the variable being read. Also, if `rich_lvl` is provided, `_R_rich_lvl` is appended to the head name if it is >0.

Furthermore, using `lim_loc`, a hyperslab of the variable can be read.

Note

If a limit of lim_loc is a negative value, the procedure just takes the entire range. This is necessary as sometimes the calling procuderes don't have, and don't need to have, knowledge of the underlying sizes, for example in the case of having multiple parallel jobs.

Returns

ierr

Parameters

in,out	*var*	variable to read
in	*pb3d_name*	name of PB3D file
in	*head_name*	head name of variables
in	*var_name*	name of variable
in	*rich_lvl*	Richardson level to reconstruct
in	*disp_info*	display additional information about variable being read
in	*lim_loc*	local limits

Definition at line 1528 of file HDF5_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.14.3 Variable Documentation

B.14.3.1 debug_hdf5_ops

```
logical, public hdf5_ops::debug_hdf5_ops = .false.
```

set to true to debug general information

Note

Debug version only

Definition at line 47 of file HDF5_ops.f90.

B.14.3.2 debug_print_hdf5_arrs

```
logical, public hdf5_ops::debug_print_hdf5_arrs = .false.
```

set to true to debug print_HDF5_arrs

Note

Debug version only

Definition at line 48 of file HDF5_ops.f90.

B.15 hdf5_utilities Module Reference

Utilities pertaining to HDF5 and XDMF.

Functions/Subroutines

- integer function, public set_1d_vars (lim_tot, lim_loc, space_id, c_plist_id)

  Sets up the chunk property list and/or the 1D hyperslabs that correspond to a local subset of `lim_tot` given by the limits `lim_loc`.
- integer function, public probe_hdf5_group (hdf5_name, group_name, group_exists)

  Probe HDF5 file for group existence.
- integer function, public list_all_vars_in_group (group_id)

  Lists all variables in a HDF5 group.

Variables

- logical, public debug_set_1d_vars = .false.

  set to true to debug set_1D_vars

B.15.1 Detailed Description

Utilities pertaining to HDF5 and XDMF.

See also

See hdf5_ops for notes on HDF5 and XDMF.

B.15.2 Function/Subroutine Documentation

B.15.2.1 list_all_vars_in_group()

```
integer function, public hdf5_utilities::list_all_vars_in_group (
            integer(hid_t), intent(in) group_id)
```

Lists all variables in a HDF5 group.

Note

Debug version only

Returns

ierr

Parameters

in	*group_id*	group identifier

Definition at line 318 of file HDF5_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.15.2.2 probe_hdf5_group()

```
integer function, public hdf5_utilities::probe_hdf5_group (
            character(len=*), intent(in) hdf5_name,
            character(len=*), intent(in) group_name,
            logical, intent(inout) group_exists)
```

Probe HDF5 file for group existence.

Returns

ierr

Parameters

in	*hdf5_name*	name of HDF5 file
in	*group_name*	name of group to probe for
in,out	*group_exists*	whether group exists

Definition at line 249 of file HDF5_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.15.2.3 set_1d_vars()

```
integer function, public hdf5_utilities::set_1d_vars (
            integer, dimension(:,:), intent(in) lim_tot,
            integer, dimension(:,:), intent(in) lim_loc,
            integer(hid_t), intent(in), optional space_id,
            integer(hid_t), intent(inout), optional c_plist_id)
```

Sets up the chunk property list and/or the 1D hyperslabs that correspond to a local subset of `lim_tot` given by the limits `lim_loc`.

An example for the hyperslab is given in 3 dimensions with size n = 5,3,3:

- for dim = 1, limits: 2..4
- for dim = 2, limits: 3..3
- for dim = 3, limits: 2..3.

The 1D equivalent can be represented as
```
[.....][.....][.....] | [.....][.....][.....] | [.....][.....][.....]
```

Now, dimension 1 will only allow for the following elements (given by -):
```
[.---.][.---.][.---.] | [.---.][.---.][.---.] | [.---.][.---.][.---.]
```

dimension 2 will only allow for the following elements:
```
[.....][.....][-----] | [.....][.....][-----] | [.....][.....][-----]
```

dimension 3 will only allow for the following elements:
```
[.....][.....][.....] | [-----][-----][-----] | [-----][-----][-----]
```

so that the total is given by:
```
[.....][.....][.....] | [.....][.....][.....] | [.....][.....][.---.]
```

In practice, it is easiest to work with a full selection, where for every dimension the ranges that do not correspond to the range of that dimension are taken away. Clearly, if a dimension has the full range, nothing will be taken out:

- `block_i` = $n_1 n_2 \cdots n_{i-1}(b_i - a_{i+1})$,
- `stride_i` = $n_1 n_2 \cdots n_{i-1}(B_i - A_{i+1})$,
- `offset_i` = $n_1 n_2 \cdots n_{i-1}(a_i - A_i)$,
- `count_i` = $n_{i+1} n_{i+2} \cdots n_N$,

where a_i and b_i represent the local limits of dimension i, A_i and B_i the total ones and the number of dimensions is N, as can be verified.

The chunk property list for creation can be set up so that its size is equal to the largest dimensions of full range, or an integer part of that if it exceeds 4GB. Since the variables don't need to be used more than once, either `nbytes` or `nslots` could be set to 0 in an access property list, but this is currently not done.

Returns

ierr

Parameters

in	*lim_tot*	total limits
in	*lim_loc*	local limits
in	*space_id*	dataspace identifier
in,out	*c_plist_id*	chunk creation property list identifier

Definition at line 79 of file HDF5_utilities.f90.

Here is the caller graph for this function:

B.15.3 Variable Documentation

B.15.3.1 debug_set_1d_vars

```
logical, public hdf5_utilities::debug_set_1d_vars = .false.
```

set to true to debug set_1D_vars

Note

Debug version only

Definition at line 24 of file HDF5_utilities.f90.

B.16 hdf5_vars Module Reference

Variables pertaining to HDF5 and XDMF.

Interfaces and Types

- type xml_str_type

  XML strings used in XDMF.
- type hdf5_file_type

  HDF5 data type, containing the information about HDF5 files.
- type var_1d_type

  1D equivalent of multidimensional variables, used for internal HDF5 storage.
- interface dealloc_xml_str

  Deallocates XML_str_type.
- interface dealloc_var_1d

  Deallocates 1D variable.

Functions/Subroutines

- subroutine, public init_hdf5

  Initializes the HDF5 types.

Variables

- character(len=6), public xmf_fmt = '(999A)'

  format to write the xmf file
- integer, parameter, public max_dim_var_1d = 100000

  maximum dimension of var_1D
- character(len=max_str_ln), dimension(2), public xdmf_num_types

  possible XDMF number types
- character(len=max_str_ln), dimension(2), public xdmf_format_types

  possible XDMF format types
- character(len=max_str_ln), dimension(2), public xdmf_geom_types

  possible XDMF geometry types
- character(len=max_str_ln), dimension(2), public xdmf_top_types

  possible XDMF topology types
- character(len=max_str_ln), dimension(2), public xdmf_att_types

  possible XDMF attribute types
- character(len=max_str_ln), dimension(2), public xdmf_center_types

  possible XDMF attribute center types
- character(len=max_str_ln), dimension(3), public xdmf_grid_types

  possible XDMF grid types

B.16.1 Detailed Description

Variables pertaining to HDF5 and XDMF.

B.16.2 Function/Subroutine Documentation

B.16.2.1 init_hdf5()

```
subroutine, public hdf5_vars::init_hdf5
```

Initializes the HDF5 types.

Has to be called once.

Definition at line 81 of file HDF5_vars.f90.

Here is the caller graph for this function:

B.16.3 Variable Documentation

B.16.3.1 max_dim_var_1d

```
integer, parameter, public hdf5_vars::max_dim_var_1d = 100000
```

maximum dimension of var_1D

Definition at line 21 of file HDF5_vars.f90.

B.16.3.2 xdmf_att_types

```
character(len=max_str_ln), dimension(2), public hdf5_vars::xdmf_att_types
```

possible XDMF attribute types

Definition at line 28 of file HDF5_vars.f90.

B.16.3.3 xdmf_center_types

```
character(len=max_str_ln), dimension(2), public hdf5_vars::xdmf_center_types
```

possible XDMF attribute center types

Definition at line 29 of file HDF5_vars.f90.

B.16.3.4 xdmf_format_types

```
character(len=max_str_ln), dimension(2), public hdf5_vars::xdmf_format_types
```

possible XDMF format types

Definition at line 25 of file HDF5_vars.f90.

B.16.3.5 xdmf_geom_types

```
character(len=max_str_ln), dimension(2), public hdf5_vars::xdmf_geom_types
```

possible XDMF geometry types

Definition at line 26 of file HDF5_vars.f90.

B.16.3.6 xdmf_grid_types

```
character(len=max_str_ln), dimension(3), public hdf5_vars::xdmf_grid_types
```

possible XDMF grid types

Definition at line 30 of file HDF5_vars.f90.

B.16.3.7 xdmf_num_types

```
character(len=max_str_ln), dimension(2), public hdf5_vars::xdmf_num_types
```

possible XDMF number types

Definition at line 24 of file HDF5_vars.f90.

B.16.3.8 xdmf_top_types

```
character(len=max_str_ln), dimension(2), public hdf5_vars::xdmf_top_types
```

possible XDMF topology types

Definition at line 27 of file HDF5_vars.f90.

B.16.3.9 xmf_fmt

```
character(len=6), public hdf5_vars::xmf_fmt = '(999A)'
```

format to write the xmf file

Definition at line 20 of file HDF5_vars.f90.

B.17 helena_ops Module Reference

Operations on HELENA variables.

Functions/Subroutines

- integer function, public read_hel (n_r_in, use_pol_flux_h)

  Reads the HELENA equilibrium data.
- integer function, public interp_hel_on_grid (grid_in, grid_out, eq_2, x_1, x_2, eq_2_out, x_1_out, x_2_↩ out, eq_1, grid_name)

  Interpolate variables resulting from HELENA equilibria to another grid (angularly).
- integer function, public test_metrics_h ()

  Checks whether the metric elements provided by HELENA are consistent with a direct calculation using the coordinate transformations [15].
- integer function, public test_harm_cont_h ()

  Investaige harmonic content of the HELENA variables.

B.17.1 Detailed Description

Operations on HELENA variables.

B.17.2 Function/Subroutine Documentation

B.17.2.1 interp_hel_on_grid()

```
integer function, public helena_ops::interp_hel_on_grid (
                type(grid_type), intent(in) grid_in,
                type(grid_type), intent(in) grid_out,
                type(eq_2_type), intent(inout), optional eq_2,
                type(x_1_type), intent(inout), optional x_1,
                type(x_2_type), intent(inout), optional x_2,
                type(eq_2_type), intent(inout), optional eq_2_out,
                type(x_1_type), intent(inout), optional x_1_out,
                type(x_2_type), intent(inout), optional x_2_out,
                type(eq_1_type), intent(in), optional eq_1,
                character(len=*), intent(in), optional grid_name)
```

Interpolate variables resulting from HELENA equilibria to another grid (angularly).

The input and output grid to be provided depend on the quantities to be interpolated:

- equilibrium variables: flux variables (no need to convert) and derived quantities (need equilibrium grid)
- metric variables: jac_FD (need equilibrium grid)
- vectorial perturbation variables: U_i, DU_i (need perturbation grid)
- tensorial perturbation variables: PV_i, KV_i (need perturbation grid)

Also, a message can be printed if a grid name is passed.

Note

1. The metric coefficients are interpolated and then compensated for the straight-field-line coordinates as in [15] .
2. By default the interpolated quantities overwrite the original ones, but alternative output variables can be provided.
3. as the equilibrium and perturbation grid are not generally identical, this routine has to be called separately for the variables tabulated in either grid.

Returns

ierr

Parameters

`in`	*grid_in*	input grid
`in`	*grid_out*	output grid
`in,out`	*eq_2*	general metric equilibrium variables
`in,out`	*x_1*	general vectorial perturbation variables
`in,out`	*x_2*	general tensorial perturbation variables
`in,out`	*eq_2_out*	field-aligned metric equilibrium variables
`in,out`	*x_1_out*	field-aligned vectorial perturbation variables
`in,out`	*x_2_out*	field-aligned tensorial perturbation variables
`in`	*eq_1*	general flux equilibrium variables for metric interpolation
`in`	*grid_name*	name of grid to which to adapt quantities

Definition at line 548 of file HELENA_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.17.2.2 read_hel()

```
integer function, public helena_ops::read_hel (
                integer, intent(inout) n_r_in,
                logical, intent(inout) use_pol_flux_h)
```

Reads the HELENA equilibrium data.

Adapted from HELENA routine `IODSK`.

The variales in the HELENA mapping file are globalized in two ways:

- X and Y are normalized w.r.t. vacuum geometric axis `R_vac` and toroidal field at the geometric axis `B_vac`.
- `R[m] = R_vac[m] (1 + eps X[]) ,`
- `Z[m] = R_vac[m] eps Y[] .`

The covariant toroidal field `F_H`, `pres_H` and poloidal flux are normalized w.r.t magnetic axis `R_m` and total toroidal field at magnetic axis `B_m`:

- `RBphi[Tm] = F_H[] R_m[m] B_m[T] ,`
- `pres[N/m^2] = pres_H[] (B_m[T])^2/mu_0[N/A^2] ,`
- `flux_p[Tm^2] = 2pi (s[])^2 cpsurf[] B_m[T] (R_m[m])^2 .`

The first normalization type is the HELENA normalization, whereas the second is the MISHKA normalization.

Everything is translated to MISHKA normalization to make comparison with MISHKA simple. This is done using the factors:

- `radius[] = a[m] / R_m[m]` ,
- `eps[] = a[m] / R_vac[m]` , so that the expressions become:
- `R[m] = radius[] (1/eps[] + X[]) R_m[m]`,
- `Z[m] = radius[] Y[] R_m[m]`,
- `RBphi[Tm] = F_H[] B_m[T] R_m[m]`,
- `pres[N/m^2] = pres_H[] B_m[T]^2 mu_0[N/A^2]^-1`,
- `flux_p[Tm^2] = 2pi (s[])^2 cpsurf[] B_m[T] R_m[T]^2`.

Finally, in HELENA, the total current `I`, the poloidal beta and the density at the geometric axis can be prescribed through:

- `XIAB = mu_0 I / (a_vac B_vac)` ,
- `BETAP = 8 pi S <p> / (I^2 mu_0)` ,
- `ZN` ,

where `a_vac = eps R_vac` and `B_vac` are vacuum quantities, `S` is the 2-D cross-sectional area and <p> is the 2-D averaged pressure.

Note

To translate this to the MISHKA normalization factors as

- `R_m = (eps/radius) R_vac` ,
- `B_m = B_vac / B0` ,

where

- `radius` is in the mapping file (12), as well as in the HELENA output (20).
- `eps` is in the mapping file (12), as well as in the HELENA input (10) and output (20).
- `B0` is in the mapping file (12), as well as in the HELENA output (20).

Furthermore, the density on axis can be specified as `ZN0` from HELENA input (10).
The other variables should probably not be touched for consistency.

Finally, the variables IAS and B0 are in the mapping file (12) only in patched versions. See Getting the Equilibrium.

Returns

ierr

Parameters

`in,out`	*n_r_in*	nr. of normal points in input grid

in,out	*use_pol_flux↩ _h*	.true. if HELENA equilibrium is based on pol. flux

Definition at line 83 of file HELENA_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.17.2.3 test_harm_cont_h()

```
integer function, public helena_ops::test_harm_cont_h
```

Investaige harmonic content of the HELENA variables.

Note

Run with one process.

Definition at line 1484 of file HELENA_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.17.2.4 test_metrics_h()

```
integer function, public helena_ops::test_metrics_h
```

Checks whether the metric elements provided by HELENA are consistent with a direct calculation using the coordinate transformations [15].

Direct calculations used:

$$
\begin{aligned}
|\nabla\psi|^2 &= \frac{1}{\mathcal{J}^2}\left(\left(\frac{\partial Z}{\partial\chi}\right)^2 + \left(\frac{\partial R}{\partial\chi}\right)^2\right) \\
|\nabla\psi\cdot\nabla\chi| &= \frac{1}{\mathcal{J}^2}\left(\frac{\partial Z}{\partial\chi}\frac{\partial Z}{\partial\psi} - \frac{\partial R}{\partial\chi}\frac{\partial R}{\partial\psi}\right) \\
|\nabla\chi|^2 &= \frac{1}{\mathcal{J}^2}\left(\left(\frac{\partial Z}{\partial\psi}\right)^2 + \left(\frac{\partial R}{\partial\psi}\right)^2\right) \\
|\nabla\phi|^2 &= \frac{1}{R^2}
\end{aligned}
$$

with

$$
\mathcal{J} = \frac{\partial Z}{\partial\psi}\frac{\partial R}{\partial\chi} - \frac{\partial R}{\partial\psi}\frac{\partial Z}{\partial\chi}
$$

Also, test whether the pressure balance $\nabla p = \vec{J}\times\vec{B}$ is satisfied.

Note

Debug version only

Returns

ierr

Definition at line 1288 of file HELENA_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.18 helena_vars Module Reference

Variables that have to do with HELENA quantities.

Functions/Subroutines

- subroutine, public dealloc_hel

  Deallocates HELENA quantities that are not used any more.

Variables

- real(dp), public rmtog_h

  R_geo/R_mag.
- real(dp), public bmtog_h

  B_geo/B_mag.
- real(dp), dimension(:), allocatable, public chi_h

  poloidal angle
- real(dp), dimension(:,:), allocatable, public flux_p_h

  poloidal flux
- real(dp), dimension(:,:), allocatable, public flux_t_h

  toroidal flux
- real(dp), dimension(:,:), allocatable, public pres_h

  pressure profile

- real(dp), dimension(:,:), allocatable, public q_saf_h

  safety factor
- real(dp), dimension(:,:), allocatable, public rot_t_h

  rotational transform
- real(dp), dimension(:,:), allocatable, public rbphi_h

  $RB_\phi(=F)$
- real(dp), dimension(:,:), allocatable, public h_h_11

  upper metric factor h_{11} (gem11)
- real(dp), dimension(:,:), allocatable, public h_h_12

  upper metric factor h_{12} (gem12)
- real(dp), dimension(:,:), allocatable, public h_h_33

  upper metric factor h_{32} (1 / gem12)
- real(dp), dimension(:,:), allocatable, public r_h

  major radius R (xout)
- real(dp), dimension(:,:), allocatable, public z_h

  height Z (yout)
- integer, public nchi

  nr. of poloidal points
- integer, public ias

  0 if top-bottom symmetric, 1 if not

B.18.1 Detailed Description

Variables that have to do with HELENA quantities.

B.18.2 Function/Subroutine Documentation

B.18.2.1 dealloc_hel()

```
subroutine, public helena_vars::dealloc_hel
```

Deallocates HELENA quantities that are not used any more.

Definition at line 40 of file HELENA_vars.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.18.3 Variable Documentation

B.18.3.1 bmtog_h

```
real(dp), public helena_vars::bmtog_h
```

B_geo/B_mag.

Definition at line 22 of file HELENA_vars.f90.

B.18.3.2 chi_h

```
real(dp), dimension(:), allocatable, public helena_vars::chi_h
```

poloidal angle

Definition at line 23 of file HELENA_vars.f90.

B.18.3.3 flux_p_h

```
real(dp), dimension(:,:), allocatable, public helena_vars::flux_p_h
```

poloidal flux

Definition at line 24 of file HELENA_vars.f90.

B.18.3.4 flux_t_h

```
real(dp), dimension(:,:), allocatable, public helena_vars::flux_t_h
```

toroidal flux

Definition at line 25 of file HELENA_vars.f90.

B.18.3.5 h_h_11

```
real(dp), dimension(:,:), allocatable, public helena_vars::h_h_11
```

upper metric factor h_{11} (gem11)

Definition at line 30 of file HELENA_vars.f90.

B.18.3.6 h_h_12

```
real(dp), dimension(:,:), allocatable, public helena_vars::h_h_12
```

upper metric factor h_{12} (gem12)

Definition at line 31 of file HELENA_vars.f90.

B.18.3.7 h_h_33

```
real(dp), dimension(:,:), allocatable, public helena_vars::h_h_33
```

upper metric factor h_{32} (1 / gem12)

Definition at line 32 of file HELENA_vars.f90.

B.18.3.8 ias

```
integer, public helena_vars::ias
```

0 if top-bottom symmetric, 1 if not

Definition at line 36 of file HELENA_vars.f90.

B.18.3.9 nchi

```
integer, public helena_vars::nchi
```

nr. of poloidal points

Definition at line 35 of file HELENA_vars.f90.

B.18.3.10 pres_h

```
real(dp), dimension(:,:), allocatable, public helena_vars::pres_h
```

pressure profile

Definition at line 26 of file HELENA_vars.f90.

B.18.3.11 q_saf_h

```
real(dp), dimension(:,:), allocatable, public helena_vars::q_saf_h
```

safety factor

Definition at line 27 of file HELENA_vars.f90.

B.18.3.12 r_h

```
real(dp), dimension(:,:), allocatable, public helena_vars::r_h
```

major radius R (xout)

Definition at line 33 of file HELENA_vars.f90.

B.18.3.13 rbphi_h

```
real(dp), dimension(:,:), allocatable, public helena_vars::rbphi_h
```

$RB_\phi(=F)$

Definition at line 29 of file HELENA_vars.f90.

B.18.3.14 rmtog_h

```
real(dp), public helena_vars::rmtog_h
```

R_geo/R_mag.

Definition at line 21 of file HELENA_vars.f90.

B.18.3.15 rot_t_h

```
real(dp), dimension(:,:), allocatable, public helena_vars::rot_t_h
```

rotational transform

Definition at line 28 of file HELENA_vars.f90.

B.18.3.16 z_h

```
real(dp), dimension(:,:), allocatable, public helena_vars::z_h
```

height Z (yout)

Definition at line 34 of file HELENA_vars.f90.

B.19 input_ops Module Reference

Operations concerning giving input.

Functions/Subroutines

- integer function, public read_input_opts ()

  Reads input options from user-provided input file.
- integer function, public read_input_eq ()

  Reads the equilibrium input file if no Richardson restart.
- integer function, public print_output_in (data_name, remove_previous_arrs)

  Print input quantities to an output file.

B.19.1 Detailed Description

Operations concerning giving input.

B.19.2 Function/Subroutine Documentation

B.19.2.1 print_output_in()

```
integer function, public input_ops::print_output_in (
                character(len=*), intent(in) data_name,
                logical, intent(in), optional remove_previous_arrs)
```

Print input quantities to an output file.

Variables printed:

- `misc_in: prog_version`, `eq_style`, `rho_style`, `R_0`, `pres_0`, `B_0`, `psi_0 rho_0`, `T_0`, `vac_perm`, `use_pol_flux_F`, `use_pol_flux_E`, `use_normalization`, `norm_disc_prec_eq`, `norm_disc_style_sol`, `n_r_in`, `n_r_eq`, `n_r_sol`, `debug_version`
- `misc_in_V: is_asym_V`, `is_freeb_V`, `mnmax_V`, `mpol_V`, `ntor_V`, `gam_V`
- `flux_t_V`
- `flux_p_V`
- `pres_V`
- `rot_t_V`
- `q_saf_V`
- `mn_V`
- `R_V_c`
- `R_V_s`
- `Z_V_c`

- Z_V_s
- L_V_c
- L_V_s
- B_V_sub: B_V_sub_c, B_V_sub_s
- B_V: B_V_c, B_V_s
- J_V_sup_int
- jac_V: jac_V_c, jac_V_s
- misc_in_H: ias, nchi
- RZ_H: R_H, Z_H
- chi_H
- q_saf_H
- rot_t_H
- RBphi_H
- pres_H
- flux_p_H
- flux_t_H
- misc_X: prim_X, n_mod_X, min_sec_X, max_sec_X, norm_disc_prec_X, norm_style, U_style, X_style, matrix_SLEPC_style, K_style, alpha_style
- misc_sol: min_r_sol, max_r_sol, norm_disc_prec_sol, BC_style, EV_BC, EV_BC

Returns

ierr

Parameters

in	*data_name*	name under which to store
in	*remove_previous_arrs*	remove previous variables if present

Definition at line 1173 of file input_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.19.2.2 read_input_eq()

```
integer function, public input_ops::read_input_eq
```

Reads the equilibrium input file if no Richardson restart.

These variables will be passed on through the HDF5 data system.

See also

reconstruct_pb3d_in().

Returns

ierr

Definition at line 1103 of file input_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.19.2.3 read_input_opts()

```
integer function, public input_ops::read_input_opts
```

Reads input options from user-provided input file.

This is done by the master process and these variables will then be passed on through MPI to other processes in broadcast_input_opts().

Returns

ierr

Definition at line 22 of file input_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.20 input_utilities Module Reference

Numerical utilities related to input.

Functions/Subroutines

- logical function, public get_log (yes, ind)

  Queries for a logical value yes or no, where the default answer is also to be provided.
- real(dp) function, public get_real (lim_lo, lim_hi, ind)

  Queries for user input for a real value, where allowable range can be provided as well.
- integer function, public get_int (lim_lo, lim_hi, ind)

  Queries for user input for an integer value, where allowable range can be provided as well.
- subroutine, public pause_prog (ind)

  Pauses the running of the program.
- subroutine, public dealloc_in ()

  Cleans up input from equilibrium codes.

B.20.1 Detailed Description

Numerical utilities related to input.

B.20.2 Function/Subroutine Documentation

B.20.2.1 dealloc_in()

```
subroutine, public input_utilities::dealloc_in
```

Cleans up input from equilibrium codes.

Definition at line 267 of file input_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.20.2.2 get_int()

```
integer function, public input_utilities::get_int (
                integer, intent(in), optional lim_lo,
                integer, intent(in), optional lim_hi,
                logical, intent(in), optional ind)
```

Queries for user input for an integer value, where allowable range can be provided as well.

If not called by all processes at the same time, `ind` should be set to indicate an individual operation.

Returns

output value

Parameters

in	*lim_lo*	upper and lower limit

in	*lim_↩ hi*	upper and lower limit
in	*ind*	individual pause or not

Definition at line 151 of file input_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.20.2.3 get_log()

```
logical function, public input_utilities::get_log (
                logical yes,
                logical, intent(in), optional ind)
```

Queries for a logical value yes or no, where the default answer is also to be provided.

If not called by all processes at the same time, `ind` should be set to indicate an individual operation.

Returns

output value

Parameters

	yes	default answer

in	*ind*	individual pause or not

Definition at line 21 of file input_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.20.2.4 get_real()

```
real(dp) function, public input_utilities::get_real (
            real(dp), intent(in), optional lim_lo,
            real(dp), intent(in), optional lim_hi,
            logical, intent(in), optional ind)
```

Queries for user input for a real value, where allowable range can be provided as well.

If not called by all processes at the same time, `ind` should be set to indicate an individual operation.

Returns

output value

Parameters

in	*lim_lo*	upper and lower limit

in	*lim_↩ hi*	upper and lower limit
in	*ind*	individual pause or not

Definition at line 76 of file input_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.20.2.5 pause_prog()

```
subroutine, public input_utilities::pause_prog (
              logical, intent(in), optional ind)
```

Pauses the running of the program.

If not called by all processes at the same time, `ind` should be set to indicate an individual operation.

Definition at line 225 of file input_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.21 messages Module Reference

Numerical utilities related to giving output.

Functions/Subroutines

- subroutine, public init_output ()

  Initialize the variables for the module.
- subroutine, public print_hello ()

  Prints first message.
- subroutine, public print_goodbye ()

  Prints last messag.
- subroutine, public init_time ()

  Intialize the time passed to 0.
- subroutine, public start_time ()

  Start a timer.
- subroutine, public stop_time ()

  Stop a timer.
- subroutine, public passed_time ()

  Display the time that has passed between `t1` and `t2`.
- subroutine, public print_err_msg (err_msg, routine_name)

  Prints an error message that is either user-provided, or the name of the calling routine.
- subroutine, public lvl_ud (inc)

  Increases/decreases `lvl` of output.
- subroutine, public writo (input_str, persistent, error, warning, alert)

  Write output to file identified by `output_i`.
- subroutine, public print_ar_2 (arr)

  Print an array of dimension 2 on the screen.
- subroutine, public print_ar_1 (arr)

  Print an array of dimension 1 on the screen.
- integer function, public get_mem_usage ()

  Returns the memory usage in kilobytes.

Variables

- integer, public lvl

  determines the indenting. higher `lvl` = more indenting
- character(len=2), public lvl_sep = ''

  characters that separate different levels of output
- character(len=10), public time_sep = ''

  defines the length of time part of output
- logical, public temp_output_active

  true if temporary output is to be written in `temp_output`
- character(len=max_str_ln), dimension(:), allocatable, public temp_output

  temporary output, before output file is opened

B.21.1 Detailed Description

Numerical utilities related to giving output.

B.21.2 Function/Subroutine Documentation

B.21.2.1 get_mem_usage()

```
integer function, public messages::get_mem_usage
```

Returns the memory usage in kilobytes.

Based on `http://stackoverflow.com/questions/22028571/track-memory-usage-in-fortran-90`

Note

Only works under linux.

Debug version only

Definition at line 553 of file messages.f90.

Here is the caller graph for this function:

B.21.2.2 init_output()

```
subroutine, public messages::init_output
```

Initialize the variables for the module.

Definition at line 31 of file messages.f90.

Here is the caller graph for this function:

B.21.2.3 init_time()

```
subroutine, public messages::init_time
```

Intialize the time passed to 0.

Definition at line 112 of file messages.f90.

Here is the caller graph for this function:

B.21.2.4 lvl_ud()

```
subroutine, public messages::lvl_ud (
            integer inc)
```

Increases/decreases `lvl` of output.

Name stands for level up or down.

Parameters

inc	increment of level

Definition at line 253 of file messages.f90.

Here is the call graph for this function:

B.21.2.5 passed_time()

`subroutine, public messages::passed_time`

Display the time that has passed between `t1` and `t2`.

Automatically stops time and resets everything to zero.

Definition at line 151 of file messages.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.21.2.6 print_ar_1()

```
subroutine, public messages::print_ar_1 (
            real(dp), dimension(:), intent(in) arr)
```

Print an array of dimension 1 on the screen.

Parameters

in	*arr*	array to be printed

Definition at line 486 of file messages.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.21.2.7 print_ar_2()

```
subroutine, public messages::print_ar_2 (
            real(dp), dimension(:,:), intent(in) arr)
```

Print an array of dimension 2 on the screen.

Parameters

in	*arr*	array to be printed

Definition at line 474 of file messages.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.21.2.8 print_err_msg()

```
subroutine, public messages::print_err_msg (
            character(len=*), intent(in) err_msg,
            character(len=*), intent(in) routine_name)
```

Prints an error message that is either user-provided, or the name of the calling routine.

Note

This should be used with the macro CHCKERR.

Parameters

in	*err_msg*	error message to be printed
in	*routine_name*	name of the routine

Definition at line 233 of file messages.f90.

Here is the call graph for this function:

B.21.2.9 print_goodbye()

```
subroutine, public messages::print_goodbye
```

Prints last messag.

Definition at line 98 of file messages.f90.

Here is the caller graph for this function:

B.21.2.10 print_hello()

```
subroutine, public messages::print_hello
```

Prints first message.

Definition at line 60 of file messages.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.21.2.11 start_time()

```
subroutine, public messages::start_time
```

Start a timer.

Definition at line 120 of file messages.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.21.2.12 stop_time()

```
subroutine, public messages::stop_time
```

Stop a timer.

Definition at line 131 of file messages.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.21.2.13 writo()

```
subroutine, public messages::writo (
            character(len=*), intent(in) input_str,
            logical, intent(in), optional persistent,
            logical, intent(in), optional error,
            logical, intent(in), optional warning,
            logical, intent(in), optional alert)
```

Write output to file identified by `output_i`.

This is done using the correct indentation for the level (`lvl_loc`) of the output.

By default, only the master outputs, but this can be changed using `persistent`.

Optionally, special formatting for error, warning or alert can be chosen.

Parameters

in	*input_str*	the name that is searched for
in	*persistent*	output even if not group master
in	*error*	error message
in	*warning*	warning message

in	*alert*	alert message

Definition at line 274 of file messages.f90.

Here is the call graph for this function:

B.21.3 Variable Documentation

B.21.3.1 lvl

```
integer, public messages::lvl
```

determines the indenting. higher `lvl` = more indenting

Definition at line 20 of file messages.f90.

B.21.3.2 lvl_sep

```
character(len=2), public messages::lvl_sep = ''
```

characters that separate different levels of output

Definition at line 21 of file messages.f90.

B.21.3.3 temp_output

```
character(len=max_str_ln), dimension(:), allocatable, public messages::temp_output
```

temporary output, before output file is opened

Definition at line 27 of file messages.f90.

B.21.3.4 temp_output_active

```
logical, public messages::temp_output_active
```

true if temporary output is to be written in `temp_output`

Definition at line 26 of file messages.f90.

B.21.3.5 time_sep

```
character(len=10), public messages::time_sep = ''
```

defines the length of time part of output

Definition at line 22 of file messages.f90.

B.22 mpi_ops Module Reference

Operations related to MPI.

Functions/Subroutines

- integer function, public start_mpi ()

  Start MPI and gather information.
- integer function, public stop_mpi (grid_eq, grid_eq_b, grid_x, grid_x_b, grid_sol, eq_1, eq_2, x_1, x_2, vac, sol)

  Stop MPI.
- integer function, public abort_mpi ()

  Abort MPI suddenly.
- integer function, public broadcast_input_opts ()

  Broadcasts options (e.g. user-prescribed) that are not passed through the HDF5 output file (i.e. `ltest`, `no_plots`, ...).
- subroutine, public sudden_stop (ierr)

  Suddenly stops the computations, aborting MPI, etc.

B.22.1 Detailed Description

Operations related to MPI.

B.22.2 Function/Subroutine Documentation

B.22.2.1 abort_mpi()

```
integer function, public mpi_ops::abort_mpi
```

Abort MPI suddenly.

Returns

ierr

Definition at line 226 of file MPI_ops.f90.

Here is the caller graph for this function:

B.22.2.2 broadcast_input_opts()

```
integer function, public mpi_ops::broadcast_input_opts
```

Broadcasts options (e.g. user-prescribed) that are not passed through the HDF5 output file (i.e. `ltest`, `no_plots`, ...).

See also

read_input_opts()

Note

Some variables (e.g. `eq_style`, ...) are not passed over MPI. Every process should call its own reconstruct_pb3d_in() in order to obtain them.

Returns

ierr

Definition at line 246 of file MPI_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.22.2.3 start_mpi()

```
integer function, public mpi_ops::start_mpi
```

Start MPI and gather information.

Returns

ierr

Definition at line 21 of file MPI_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.22.2.4 stop_mpi()

```
integer function, public mpi_ops::stop_mpi (
                type(grid_type), intent(inout), optional grid_eq,
                type(grid_type), intent(inout), optional, pointer grid_eq_b,
                type(grid_type), intent(inout), optional grid_x,
                type(grid_type), intent(inout), optional, pointer grid_x_b,
                type(grid_type), intent(inout), optional grid_sol,
                type(eq_1_type), intent(inout), optional eq_1,
                type(eq_2_type), intent(inout), optional eq_2,
                type(x_1_type), intent(inout), optional x_1,
                type(x_2_type), intent(inout), optional x_2,
                type(vac_type), intent(inout), optional vac,
                type(sol_type), intent(inout), optional sol)
```

Stop MPI.

Also deallocates:

- grid_eq
- grid_eq_B
- grid_X
- grid_X_B
- grid_sol
- eq_1
- eq_2
- X_1
- X_2
- sol

Returns

ierr

Parameters

in,out	*grid_eq*	equilibrium grid

in,out	*grid_eq↩ _b*	field-aligned equilibrium grid
in,out	*grid_x*	perturbation grid
in,out	*grid_x_b*	field-aligned perturbation grid
in,out	*grid_sol*	solution grid
in,out	*eq_1*	Flux equilibrium variables
in,out	*eq_2*	metric equilibrium variables
in,out	*x_1*	vectorial perturbation variables
in,out	*x_2*	integrated tensorial perturbation variables
in,out	*vac*	vacuum variables
in,out	*sol*	solution variables

Definition at line 79 of file MPI_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.22.2.5 sudden_stop()

```
subroutine, public mpi_ops::sudden_stop (
            integer, intent(in) ierr)
```

Suddenly stops the computations, aborting MPI, etc.

As a special case, if `ierr` = 66, no error message is printed.

Returns

ierr

Definition at line 482 of file MPI_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.23 mpi_utilities Module Reference

Numerical utilities related to MPI.

Interfaces and Types

- interface get_ser_var

  Gather parallel variable in serial version on group master.
- interface get_ghost_arr

  Fill the ghost regions in an array.
- interface broadcast_var

  Wrapper function to broadcast a single variable using MPI.

Functions/Subroutines

- integer function, public redistribute_var (var, dis_var, lims, lims_dis)

  Redistribute variables according to new limits.
- integer function, public wait_mpi ()

  Wait for all processes, wrapper to MPI barrier.
- integer function, public lock_req_acc (lock, blocking)

  Request access to lock of a BL (blocking) or optionally a NB (non-blocking) type.
- integer function, public lock_return_acc (lock)

  Returns access to a lock.
- integer function, public lock_wl_change (wl_action, blocking, lock_loc, wl, ranks)

  Adds, removes or sets to active a rank from the waiting list for a lock and returns the lock waiting list:
- character(len=max_str_ln) function, public lock_header (lock_loc)

  Returns the header for lock debug messages.

Variables

- logical, public debug_lock = .false.

  print debug information about lock operations
- integer, public n_waits = 0

  number of waits

B.23.1 Detailed Description

Numerical utilities related to MPI.

This includes a lock system, which can be both BL (blocking) or NB (non-blocking). It is based on the implementation of an MPI-IO atomic mode without file support, described in [13].

See also

See mpi_vars.

The reason for this was the fact that using a simple lock file can lead to crashes.

Note

A downside of this method is that in some rare cases a deadlock may occur as the master process, which contains the shared variable with a window that other processes may use, is idle and waiting, whereas the others are still performing lock options. As the master is idle and waiting, its MPI asynchronous communication is not performed. To remedy this, just call wait_MPI() after procedures where lock operations are performed.

B.23.2 Function/Subroutine Documentation

B.23.2.1 lock_header()

```
character(len=max_str_ln) function, public mpi_utilities::lock_header (
            type(lock_type), intent(in) lock_loc)
```

Returns the header for lock debug messages.

Note

Debug version only

Parameters

in	*lock_loc*	lock

Definition at line 1216 of file MPI_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.23.2.2 lock_req_acc()

```
integer function, public mpi_utilities::lock_req_acc (
            type(lock_type), intent(inout) lock,
            logical, intent(in), optional blocking)
```

Request access to lock of a BL (blocking) or optionally a NB (non-blocking) type.

Note

Based on [13].

Returns

ierr

Definition at line 764 of file MPI_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.23.2.3 lock_return_acc()

```
integer function, public mpi_utilities::lock_return_acc (
            type(lock_type), intent(inout) lock)
```

Returns access to a lock.

The blocking property has been set when requesting the lock.

Note

Based on [13].

Returns

ierr

Definition at line 871 of file MPI_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.23.2.4 lock_wl_change()

```
integer function, public mpi_utilities::lock_wl_change (
                integer, intent(in) wl_action,
                logical, intent(in) blocking,
                type(lock_type), intent(inout) lock_loc,
                integer, dimension(:), intent(inout), allocatable wl,
                integer, dimension(:), intent(in), optional ranks)
```

Adds, removes or sets to active a rank from the waiting list for a lock and returns the lock waiting list:

Actions:

- `wl_action` = 0: remove
- `wl_action` = 1: add
- `wl_action` = 2: active

Or negative equivalents for non-blocking (NB) procs.

Optionally, the rank(s) of the process for which to perform this action can be passed. This is useful for doing the same action on multiple processes.

Note

Based on [13].
Debug version only

Returns

ierr

Parameters

in	*wl_action*	action to perform
in	*blocking*	the ranks to be changed are blocking
in,out	*lock_loc*	lock
in,out	*wl*	waiting list
in	*ranks*	rank(s) for which to perform option

Definition at line 1109 of file MPI_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.23.2.5 redistribute_var()

```
integer function, public mpi_utilities::redistribute_var (
            real(dp), dimension(:), intent(in) var,
            real(dp), dimension(:), intent(inout) dis_var,
            integer, dimension(2), intent(in) lims,
            integer, dimension(2), intent(in) lims_dis)
```

Redistribute variables according to new limits.

Parameters

in	*var*	parallel vector
in,out	*dis_var*	redistributed vector
in	*lims*	indices of parallel vector
in	*lims_dis*	indices of redistributed parallel vector

Definition at line 329 of file MPI_utilities.f90.

Here is the caller graph for this function:

B.23.2.6 wait_mpi()

```
integer function, public mpi_utilities::wait_mpi
```

Wait for all processes, wrapper to MPI barrier.

Returns

ierr

Definition at line 743 of file MPI_utilities.f90.

Here is the caller graph for this function:

B.23.3 Variable Documentation

B.23.3.1 debug_lock

```
logical, public mpi_utilities::debug_lock = .false.
```

print debug information about lock operations

Note

Debug version only

Definition at line 40 of file MPI_utilities.f90.

B.23.3.2 n_waits

```
integer, public mpi_utilities::n_waits = 0
```

number of waits

Note

Debug version only

Definition at line 41 of file MPI_utilities.f90.

B.24 mpi_vars Module Reference

Variables pertaining to MPI.

Interfaces and Types

- type lock_type
  lock type

Functions/Subroutines

- integer function, public init_lock (lock_loc, wu_tag)
  Initializes a lock.
- integer function, public dealloc_lock (lock_loc)
  Deallocates a lock.

Variables

- type(lock_type), public hdf5_lock
  HDF5 lock.

B.24.1 Detailed Description

Variables pertaining to MPI.

B.24.2 Function/Subroutine Documentation

B.24.2.1 dealloc_lock()

```
integer function, public mpi_vars::dealloc_lock (
            class(lock_type), intent(inout) lock_loc)
```

Deallocates a lock.

Note

Should be called collectively.

Returns

ierr

Parameters

in,out	*lock_loc*	lock

Definition at line 132 of file MPI_vars.f90.

Here is the caller graph for this function:

B.24.2.2 init_lock()

```
integer function, public mpi_vars::init_lock (
            class(lock_type), intent(inout) lock_loc,
            integer, intent(in) wu_tag)
```

Initializes a lock.

Note

1. Should be called collectively.
2. Every lock should have a unique wakeup tag.

Returns

ierr

Parameters

in,out	*lock_loc*	lock

in	*wu_tag*	wakeup tag

Definition at line 86 of file MPI_vars.f90.

B.24.3 Variable Documentation

B.24.3.1 hdf5_lock

```
type(lock_type), public mpi_vars::hdf5_lock
```

HDF5 lock.

Definition at line 76 of file MPI_vars.f90.

B.25 num_ops Module Reference

Numerical operations.

Interfaces and Types

- interface calc_zero_hh
  Finds the zero of a function using Householder iteration.

Functions/Subroutines

- character(len=max_str_ln) function, public calc_zero_zhang (zero, fun, x_int_in)
  Finds the zero of a function using Zhang's method, which is simpler than Brent's method.

Variables

- logical, public debug_calc_zero = .false.
  plot debug information for calc_zero

B.25.1 Detailed Description

Numerical operations.

B.25.2 Function/Subroutine Documentation

B.25.2.1 calc_zero_zhang()

```
character(len=max_str_ln) function, public num_ops::calc_zero_zhang (
            real(dp), intent(inout) zero,
            external real(dp) function(real(dp), intent(in) x) fun,
            real(dp), dimension(2), intent(in) x_int_in)
```

Finds the zero of a function using Zhang's method, which is simpler than Brent's method.

Taken from from Steven Stage's correction of Zhang's paper [19].

Unlike Householder, Zhang's method needs an interval `x_int_in` to work in, not a guess. Also, it does not require the derivative of the function.

The routine returns an error message if no zero is found, and which is empty otherwise.

Note

The interval `x_int_in` needs to be so that the function values at either end are of different value.

Parameters

`in,out`	*fun*	`fun(x)` with • x abscissa • `fun` ordinate
`in,out`	*zero*	output
`in`	*x_int_in*	interval

Returns

possible error message

Definition at line 461 of file num_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.25.3 Variable Documentation

B.25.3.1 debug_calc_zero

```
logical, public num_ops::debug_calc_zero = .false.
```

plot debug information for calc_zero

Note

Debug version only

Definition at line 22 of file num_ops.f90.

B.26 num_utilities Module Reference

Numerical utilities.

Interfaces and Types

- interface add_arr_mult

  Add to an array (3) the product of arrays (1) and (2).
- interface calc_det

  Calculate determinant of a matrix.
- interface calc_inv

  Calculate inverse of square matrix A.
- interface calc_mult

  Calculate matrix multiplication of two square matrices $\overline{AB} = \overline{A}\,\overline{B}$.
- interface conv_mat

  Converts a (symmetric) matrix A with the storage convention described in eq_vars.eq_2_type.
- interface calc_int

  Integrates a function using the trapezoidal rule.
- interface round_with_tol

  Rounds an arry of values to limits, with a tolerance 10^{-5} *that can optionally be modified.*
- interface con2dis

  Convert between points from a continuous grid to a discrete grid.

- interface dis2con

  Convert between points from a discrete grid to a continuous grid.
- interface con

  Either takes the complex conjugate of a square matrix element A, defined on a 3-D grid, or not.
- interface bubble_sort

  Sorting with the bubble sort routine.
- interface order_per_fun

  Order a periodic function to include $0 \ldots 2\pi$ and an overlap.
- interface spline

  Wrapper to the pspline library, making it easier to use for 1-D applications where speed is not the main priority. If spline representations are to be reused, manually use the library.

Functions/Subroutines

- subroutine, public calc_aux_utilities (n_mod)

  Initialize utilities for fast future reference, depending on program style.
- integer function, dimension(:,:), allocatable, public derivs (order, dims)

  Returns derivatives of certain order.
- integer function, public check_deriv (deriv, max_deriv, sr_name)

  checks whether the derivatives requested for a certain subroutine are valid
- integer function, public calc_ext_var (ext_var, var, var_points, ext_point, deriv_in)

  Extrapolates a function.
- integer function, public calc_coeff_fin_diff (deriv, nr, ind, coeff)

  Calculate the coefficients for finite differences.
- integer function, public c (ij, sym, n, lim_n)

  Convert 2-D coordinates (i,j) to the storage convention used in matrices.
- recursive integer function, public fac (n)

  Calculate factorial.
- integer function, public is_sym (n, nn, sym)

  Determines whether a matrix making use of the storage convention in eq_vars.eq_2_type is symmetric or not.
- recursive integer function, public lcm (u, v)

  Returns common multiple using the Euclid's algorithm.
- recursive integer function, public gcd (u, v)

  Returns least denominator using the GCD.
- subroutine, public shift_f (al, bl, cl, a, b, c)

  Calculate multiplication through shifting of fourier modes A and B into C.
- subroutine, public solve_vand (n, a, b, x, transp)

  Solve a Vandermonde system $\overline{A}\,\vec{X} = \vec{B}$.
- integer function, public calc_d2_smooth (fil_n, x, y, d2y, style)

  Calculate second derivative with smoothing formula by Holoborodko, [9].

Variables

- integer, dimension(:,:,:), allocatable, public d

  1-D array indices of derivatives
- integer, dimension(:,:), allocatable, public m

  1-D array indices of metric indices
- integer, dimension(:,:), allocatable, public f

  1-D array indices of Fourier mode combination indices
- logical, public debug_con2dis_reg = .false.

  plot debug information for con2dis_reg()
- logical, public debug_calc_coeff_fin_diff = .false.

  plot debug information for calc_coeff_fin_diff()

B.26.1 Detailed Description

Numerical utilities.

B.26.2 Function/Subroutine Documentation

B.26.2.1 c()

```
integer function, public num_utilities::c (
                integer, dimension(2), intent(in) ij,
                logical, intent(in) sym,
                integer, intent(in), optional n,
                integer, dimension(2,2), intent(in), optional lim_n)
```

Convert 2-D coordinates (i,j) to the storage convention used in matrices.

Their size is by default taken to be 3:

$$\begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix} \text{ or } \begin{pmatrix} 1 & & \\ 2 & 4 & \\ 3 & 5 & 6 \end{pmatrix} \text{ for symmetic matrices .}$$

Optionally, this can be changed using n.

The value of c is then given by

$$c = (j-1)n + i$$

for non-symmetric matrices, and by

$$c = (j-1)n + i - (j-1)\frac{j}{2} \quad \text{if } i > j$$
$$c = (i-1)n + j - (i-1)\frac{i}{2} \quad \text{if } j > i$$

since $\sum_{k=1}^{j-1} k = \frac{(j-1)j}{2}$.

For local indices in submatrices, the limits in both dimensions have to be passed. The results for the full matrix are then subtracted by amount corresponding to the left, above and below parts with respect to the submatrix:

- left:
$$\sum_{i=1}^{m(2)-1} n - i + 1 = (m(2)-1)\left(n + 1 - \frac{m(2)}{2}\right)$$
- above (if positive):
$$\sum_{i=1}^{j} (m(1) - m(2) + 1 - i) = j\left(m(1) - m(2) + \frac{1}{2} - \frac{j^*}{2}\right) \text{ , with } j^* = \text{min}\,(0, j, m(1) - m(2) + 1)$$
- below:
$$\sum_{i=1}^{j-1} n - M(1) = (n - M(1))\,(j-1)$$

where m = `min` and M = `max`.

Note

1. The submatrix version is not fast, so results should be saved and reused.
2. No checks are done whether the indices make sense.

Parameters

in	*ij*	2D coords. (i,j)

in	*sym*	whether symmetric
in	*n*	max of 2-D coords.
in	*lim↩ _n*	min. and max. of 2-D coords.

Definition at line 2555 of file num_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.26.2.2 calc_aux_utilities()

```
subroutine, public num_utilities::calc_aux_utilities (
            integer, intent(in), optional n_mod)
```

Initialize utilities for fast future reference, depending on program style.

Utilities initialized:

- derivatives
- metrics
- Fourier modes (optionally)

If Fourier modes are also initialized, the quantity `n_mod` has to be provided as well.

Parameters

in	*n_mod*	n_mod for Fourier modes

Definition at line 2062 of file num_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.26.2.3 calc_coeff_fin_diff()

```
integer function, public num_utilities::calc_coeff_fin_diff (
                integer, intent(in) deriv,
                integer, intent(in) nr,
                integer, intent(in) ind,
                real(dp), dimension(:), intent(inout), allocatable coeff)
```

Calculate the coefficients for finite differences.

These represent a derivative of degree `deriv` at an index `ind`, by the weighted sum of a number `nr` of points with `1<=ind<=nr`.

They are found by solving a Vandermonde system, multiplied by the faculty of the derivative.

The result returned in `coeff(1:nr)` are used in

$$\frac{df}{dx} = \sum_{j=1}^{n} c(j) f(i+j).$$

where i = `ind`, n = `nr` and c = `coeff`

Note

They need to be divided by the step size before usage.

Examples:

- symmetric finite differences for derivative `deriv` of order `ord`:
  - $m =$ ceiling$(\frac{p+d}{2})$ to guarantee the order
  - $n = 1 + 2m$
  - $i = 1 + m$
- left finite differences for derivative `deriv` of order `ord`:
  - $n = 1 + d + p$
  - $i = m$ where p = `ord`, d = `deriv` and m = `nr_2`.

Returns

ierr

Parameters

`in`	*deriv*	degree of derivative
`in`	*nr*	number of points
`in`	*ind*	position of derivative
`in,out`	*coeff*	output coefficients

Definition at line 2448 of file num_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.26.2.4 calc_d2_smooth()

```
integer function, public num_utilities::calc_d2_smooth (
                integer, intent(in) fil_n,
                real(dp), dimension(:), intent(in) x,
                real(dp), dimension(:), intent(in) y,
                real(dp), dimension(:), intent(inout) d2y,
                integer, intent(in) style)
```

Calculate second derivative with smoothing formula by Holoborodko, [9].

The formula used is

$$s(k) = [(2N-10)\, s(k+1) - (N+2k+3)\, s(k+2)] \,/\, (N-2k-1)$$

with $s(k > M) = 0$ and $(k = N) = 1$.

for a given N.

For `style` 1, central differences are used:

$$\frac{\partial^2 y}{\partial x^2} \approx \frac{1}{2^{N-3}} \left(\sum_{k=1}^{M} \alpha_k \left(y_k + y_{-k}\right) - 2y_0 \sum_{k=1}^{M} \alpha_k \right)$$

and for `style` 2, backward differences:

$$\frac{\partial^2 y}{\partial x^2} \approx \frac{1}{2^{N-3}} \left(\sum_{k=1}^{M} \beta_k \left(y_{-M+k} + y_{-M-k}\right) - 2y_{-M} \sum_{k=1}^{M} \beta_k \right)$$

with α_k and β_k given by

$$\begin{aligned} \alpha_k &= \frac{4k^2 s_k}{\left(x_k - x_{-k}\right)^2} \\ \beta_k &= \frac{4k^2 s_k}{\left(x_{-M+k} - x_{-M-k}\right)^2} \end{aligned} .$$

Parameters

in	*fil↩ _n*	filter length (must be odd)
in	*x*	input abscissa
in	*y*	input ordinate

in,out	*d2y*	second derivative

Definition at line 2828 of file num_utilities.f90.

Here is the caller graph for this function:

B.26.2.5 calc_ext_var()

```
integer function, public num_utilities::calc_ext_var (
                real(dp), intent(inout) ext_var,
                real(dp), dimension(:), intent(in) var,
                real(dp), dimension(:), intent(in) var_points,
                real(dp), intent(in) ext_point,
                integer, intent(in), optional deriv_in)
```

Extrapolates a function.

This is done using linear or quadratic interpolation, depending on the number of points and values given.

The data should be given sorted, in ascending order, without duplicate points in var_points.

It uses the following solution:

$$\overline{\mathsf{A}}\,\vec{b} = \vec{c}$$

which is written out to

$$\begin{pmatrix} x_1^0 & x_1^1 & x_1^2 & \dots \\ x_2^0 & x_2^1 & x_2^2 & \dots \\ x_3^0 & x_3^1 & x_3^2 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \end{pmatrix} = \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ \vdots \end{pmatrix}$$

where v_{i-1} = `var_i`.

This is solved for the coefficients of the interpolating polynomial

`ext_var = a_0 + a_1 x + a_2 x^2 + ...`

There is an optional flag to look for the kth derivative of the function instead of the function itself, where k should be lower than the degree of the polynomial.

Returns

ierr

Parameters

in,out	*ext_var*	output

in	*var*	abscissa
in	*var_points*	ordinate
in	*ext_point*	point to which extrapolate
in	*deriv_in*	specifies an optional derivative

Definition at line 2323 of file num_utilities.f90.

B.26.2.6 check_deriv()

```
integer function, public num_utilities::check_deriv (
                integer, dimension(3), intent(in) deriv,
                integer, intent(in) max_deriv,
                character(len=*), intent(in) sr_name)
```

checks whether the derivatives requested for a certain subroutine are valid

Returns

ierr

Parameters

in	*deriv*	derivative to be checked
in	*max_deriv*	maximum derivative allowed
in	*sr_name*	name of subroutine

Definition at line 2259 of file num_utilities.f90.

Here is the caller graph for this function:

B.26.2.7 derivs()

```
integer function, dimension(:,:), allocatable, public num_utilities::derivs (
            integer, intent(in) order,
            integer, intent(in), optional dims)
```

Returns derivatives of certain order.

Parameters

in	*order*	order of derivative
in	*dims*	nr. of dimensions

Returns

array of all unique derivatives

Definition at line 2245 of file num_utilities.f90.

Here is the caller graph for this function:

B.26.2.8 fac()

```
recursive integer function, public num_utilities::fac (
            integer, intent(in) n)
```

Calculate factorial.

Parameters

in	*n*	input

Returns

output

Definition at line 2608 of file num_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.26.2.9 gcd()

```
recursive integer function, public num_utilities::gcd (
                integer, intent(in) u,
                integer, intent(in) v)
```

Returns least denominator using the GCD.

See also

From https://rosettacode.org/wiki/Least_common_multiple#Fortran

Parameters

in	u	input
in	v	input

Returns

result

Definition at line 2668 of file num_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.26.2.10 is_sym()

```
integer function, public num_utilities::is_sym (
            integer, intent(in) n,
            integer, intent(in) nn,
            logical, intent(inout) sym)
```

Determines whether a matrix making use of the storage convention in eq_vars.eq_2_type is symmetric or not.

Returns

ierr

Parameters

in	*n*	size of matrix
in	*nn*	number of elements in matrix
in,out	*sym*	output

Definition at line 2625 of file num_utilities.f90.

Here is the caller graph for this function:

B.26.2.11 lcm()

```
recursive integer function, public num_utilities::lcm (
            integer, intent(in) u,
            integer, intent(in) v)
```

Returns common multiple using the Euclid's algorithm.

See also

From https://rosettacode.org/wiki/Greatest_common_divisor#Recursive_Euclid_algorithm_3

Parameters

in	*u*	input
in	*v*	input

Returns

result

Definition at line 2656 of file num_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.26.2.12 shift_f()

```
subroutine, public num_utilities::shift_f (
                integer, dimension(2), intent(in) al,
                integer, dimension(2), intent(in) bl,
                integer, dimension(2), intent(in) cl,
                real(dp), dimension(al(1):al(2),2), intent(in) a,
                real(dp), dimension(bl(1):bl(2),2), intent(in) b,
                real(dp), dimension(cl(1):cl(2),2), intent(inout) c)
```

Calculate multiplication through shifting of fourier modes A and B into C.

This works by calculating $\left[\sum_A (\alpha_A \cos m_A\theta + \beta_A \sin m_A\theta)\right] \left[\sum_B (\alpha_B \cos m_B\theta + \beta_B \sin m_B\theta)\right] = \left[\sum_C (\alpha_C \cos m_C\theta + \beta_C \sin m_C\theta)\right]$

where the α and β factors are provide in A, B and C.

This then boils down to finding the four combinations of cosines and sines.

Note

Modes that are larger than what C can hold are thrown away.

Parameters

in	*al*	limits on A mode numbers

in	*bl*	limits on B mode numbers
in	*cl*	limits on C mode numbers
in	*a*	input
in	*b*	input
in,out	*c*	result

Definition at line 2695 of file num_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.26.2.13 solve_vand()

```
subroutine, public num_utilities::solve_vand (
                integer, intent(in) n,
                real(dp), dimension(n), intent(in) a,
                real(dp), dimension(n), intent(in) b,
                real(dp), dimension(n), intent(inout) x,
                logical, intent(in), optional transp)
```

Solve a Vandermonde system $\overline{\mathrm{A}}\,\vec{X} = \vec{B}$.

The Vandermonde matrix has the form

$$A = \begin{pmatrix} 1 & a_1 & a_1^2 & \cdots & a_1^n \\ 1 & a_2 & a_2^2 & \cdots & a_2^n \\ 1 & a_3 & a_3^2 & \cdots & a_3^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & a_n & a_n^2 & \cdots & a_n^n \end{pmatrix}$$

See also

Adapted from two routines dvand and pvand in https://people.sc.fsu.edu/~jburkardt/f_src/vandermonde/vandermonde.html based on the Björk-Pereyra Algorithm [2].

Parameters

in	*n*	matrix size

in	*a*	parameters of Vandermonde matrix
in	*b*	right-hand side
in,out	*x*	solution
in	*transp*	transposed Vandermonde matrix

Definition at line 2750 of file num_utilities.f90.

Here is the caller graph for this function:

B.26.3 Variable Documentation

B.26.3.1 d

```
integer, dimension(:,:,:), allocatable, public num_utilities::d
```

1-D array indices of derivatives

Definition at line 23 of file num_utilities.f90.

B.26.3.2 debug_calc_coeff_fin_diff

```
logical, public num_utilities::debug_calc_coeff_fin_diff = .false.
```

plot debug information for calc_coeff_fin_diff()

Note

Debug version only

Definition at line 28 of file num_utilities.f90.

B.26.3.3 debug_con2dis_reg

```
logical, public num_utilities::debug_con2dis_reg = .false.
```

plot debug information for con2dis_reg()

Note

Debug version only

Definition at line 27 of file num_utilities.f90.

B.26.3.4 f

```
integer, dimension(:,:), allocatable, public num_utilities::f
```

1-D array indices of Fourier mode combination indices

Definition at line 25 of file num_utilities.f90.

B.26.3.5 m

```
integer, dimension(:,:), allocatable, public num_utilities::m
```

1-D array indices of metric indices

Definition at line 24 of file num_utilities.f90.

B.27 num_vars Module Reference

Numerical variables used by most other modules.

Variables

- integer, parameter, public dp = REAL64
  double precision
- integer, parameter, public dpi = INT64
  double precision
- real(dp), parameter, public weight_dp = 0.008
  size of double precision in kB
- integer, parameter, public max_str_ln = 120
  maximum length of strings
- integer, parameter, public max_name_ln = 30
  maximum length of filenames
- integer, parameter, public max_deriv = 2
  highest derivatives for metric factors in Flux coords.
- integer, public prog_style
  program style (1: PB3D, 2: PB3D_POST)
- character(len=4), public prog_name
  name of program, used for info
- character(len=3), parameter, public output_name = 'out'
  name of output file
- character(len=14), parameter, public shell_commands_name = 'shell_commands'
  name of shell commands file
- character(len=9), parameter, public mem_usage_name = 'mem_usage'
  name of memory usage file
- integer, public mem_usage_count
  counter for memory usage output

- real(dp), parameter, public prog_version = 2.47_dp

  version number
- real(dp), parameter, public min_pb3d_version = 2.43_dp

  minimum PB3D version for POST
- logical, public debug_version = .true.

  debug version used
- integer, public rank

  MPI rank.
- integer, public n_procs

  nr. of MPI processes
- integer, public sol_n_procs

  nr. of MPI processes for solution with SLEPC
- integer(kind=8), public time_start

  start time of simulation
- real(dp), public max_tot_mem

  maximum total memory for all processes [MB]
- real(dp), public max_x_mem

  maximum memory for perturbation calculations for all processes [MB]
- integer, dimension(:,:), allocatable, public x_jobs_lims

  data about X jobs: [$\min_k$, $\max_k$, $\min_m$, $\max_m$ *] for all jobs*
- integer, dimension(:,:), allocatable, public eq_jobs_lims

  data about eq jobs: [$\min_\theta$, $\max_\theta$ *] for all jobs*
- integer, public x_job_nr

  nr. of X job
- integer, public eq_job_nr

  nr. of eq job
- real(dp), parameter, public mem_scale_fac = 6.0

  empirical scale factor of memory to calculate eq compared to just storing it
- real(dp), parameter, public pi =4_dp∗datan(1.0_dp)

  π
- real(dp), parameter, public mu_0_original = 4E-7_dp∗pi

  permeability of free space
- complex(dp), parameter, public iu = (0, 1)

  complex unit
- integer, public ev_style

  determines the method used for solving an EV problem
- integer, public eq_style

  either 1 (VMEC) or 2 (HELENA)
- integer, public rho_style

  style for equilibrium density profile
- integer, public u_style

  style for calculation of U (1: ord.2, 2: ord.1, 1: ord.0)
- integer, public norm_style

  style for normalization
- integer, public k_style

  style for kinetic energy
- integer, dimension(2), public bc_style

  style for BC left and right
- integer, public x_style

  style for secondary mode numbers (1: prescribed, 2: fast)
- integer, public matrix_slepc_style

style for matrix storage (1: sparse, 2: shell)

- integer, public solver_slepc_style

  style for solver (1: Krylov-Schur, 2: GD)
- integer, public post_style

  style for POST (1: extended grid, 2: B-aligned grid)
- integer, public x_grid_style

  style for normal component of X grid (1: eq, 2: sol, 3: enriched)
- integer, public alpha_style

  style for alpha (1: one field line, many turns, 2: many field lines, one turn)
- integer, public max_it_slepc

  maximum nr. of iterations for SLEPC
- logical, public plot_resonance

  whether to plot the q-profile or iota-profile with resonances
- logical, public plot_magn_grid

  whether to plot the grid in real coordinates
- logical, public plot_b

  whether to plot the magnetic field in real coordinates
- logical, public plot_j

  whether to plot the current in real coordinates
- logical, public plot_flux_q

  whether to plot flux quantities in real coordinates
- logical, public plot_kappa

  whether to plot curvature
- logical, public plot_sol_xi

  whether to plot plasma perturbation of solution in POST
- logical, public plot_sol_q

  whether to plot magnetic perturbation of solution in POST
- logical, public plot_vac_pot

  whether to plot vacuum potential in POST
- logical, public plot_e_rec

  whether to plot energy reconstruction in POST
- logical, public ltest

  whether or not to call the testing routines
- logical, public use_pol_flux_e

  whether poloidal flux is used in E coords.
- logical, public use_pol_flux_f

  whether poloidal flux is used in F coords.
- logical, public use_normalization

  whether to use normalization or not
- real(dp), public ev_bc

  value of artificial Eigenvalue for boundary condition
- real(dp), public ev_guess

  first guess for eigenvalue
- real(dp), dimension(:), allocatable, public tol_slepc

  tolerance for SLEPC for different Richardson levels
- real(dp), public max_njq_change

  maximum change of prim. mode number times saf. fac. / rot. transf. when using X_style 2 (fast)
- integer, public norm_disc_prec_eq

  precision for normal discretization for equilibrium
- integer, public norm_disc_prec_x

  precision for normal discretization for perturbation

- integer, public norm_disc_prec_sol

  precision for normal discretization for solution
- integer, public norm_disc_style_sol

  style for normal discretization for solution (1: central fin. diff., 2: left fin. diff.)
- integer, public magn_int_style

  style for magnetic integrals (1: trapezoidal, 2: Simpson 3/8)
- integer, public max_it_rich

  number of levels for Richardson extrapolation
- real(dp), public tol_rich

  tolerance for Richardson extrapolation
- integer, public max_it_zero

  maximum number of iterations to find zeros
- integer, public max_nr_backtracks_hh

  maximum number of backtracks for Householder, relax. factors
- real(dp), public tol_zero

  tolerance for zeros
- real(dp), public tol_norm

  tolerance for normal range (normalized to 0..1)
- integer, parameter, public ex_max_size = 10000

  maximum size of matrices for external plot
- character(len=max_str_ln), public eq_name

  name of equilibrium file from VMEC or HELENA
- character(len=max_str_ln), public pb3d_name

  name of PB3D output file
- logical, public no_plots = .false.

  no plots made
- logical, public jump_to_sol = .false.

  jump to solution
- logical, public export_hel = .false.

  export HELENA
- logical, public plot_vmec_modes = .false.

  plot VMEC modes
- logical, public invert_top_bottom_h = .false.

  invert top and bottom for HELENA equilibria
- logical, public no_output = .false.

  no output shown
- logical, public post_output_full = .false.

  POST has output on full grids.
- logical, public post_output_sol = .false.

  POST has outputs of solution.
- logical, public do_execute_command_line = .false.

  call "execute_command_line" inside program
- logical, public print_mem_usage = .false.

  print memory usage is printed
- logical, public swap_angles = .false.

  swap angles theta and zeta in plots (only for POST)
- logical, public compare_tor_pos = .false.

  compare quantities at toroidal positions (only for POST)
- logical, public retain_all_sol

  retain also faulty solutions
- character(len=5), public plot_dir = 'Plots'

directory where to save plots

- character(len=7), public script_dir = 'Scripts'

  directory where to save scripts for plots
- character(len=4), public data_dir = 'Data'

  directory where to save data for plots
- integer, public n_theta_plot

  nr. of poloidal points in plot
- integer, public n_zeta_plot

  nr. of toroidal points in plot
- integer, dimension(2), public n_vac_plot

  nr. of points in R and Z in vacuum
- real(dp), public min_theta_plot

  min. of `theta_plot`
- real(dp), public max_theta_plot

  max. of `theta_plot`
- real(dp), public min_zeta_plot

  min. of `zeta_plot`
- real(dp), public max_zeta_plot

  max. of `zeta_plot`
- real(dp), public min_r_plot

  min. of `r_plot`
- real(dp), public max_r_plot

  max. of `r_plot`
- real(dp), public min_rvac_plot

  min. of `R` in which to plot vacuum
- real(dp), public max_rvac_plot

  max. of `R` in which to plot vacuum
- real(dp), public min_zvac_plot

  min. of `Z` in which to plot vacuum
- real(dp), public max_zvac_plot

  max. of `Z` in which to plot vacuum
- integer, public plot_grid_style

  style for POST plot grid (0: 3-D plots, 1: slab plots, 2: slab plots with folding, 3: straight cylinder))
- integer, public n_sol_requested

  how many solutions requested
- integer, dimension(4), public n_sol_plotted

  how many solutions to be plot (first unstable, last unstable, first stable, last stable)
- integer, dimension(2), public plot_size

  size of plot in inches
- integer, public rich_restart_lvl

  starting Richardson level (0: none [default])
- character(len=max_str_ln), public input_name

  will hold the full name of the input file
- integer, public ex_plot_style

  external plot style (1: GNUPlot, 2: Bokeh for 2D, Mayavi for 3D)
- real(dp), public pert_mult_factor_post

  factor with which to multiply perturbation strength for POST
- real(dp), dimension(2), public rz_0

  origin of geometrical poloidal coordinate
- integer, parameter, public input_i = 50

  file number of input file

- integer, parameter, public eq_i = 51

  file number of equilibrium file from VMEC or HELENA
- integer, parameter, public pb3d_i = 52

  file number of PB3D output file
- integer, parameter, public output_i = 53

  file number of output file
- integer, parameter, public shell_commands_i = 54

  file number of shell commands file
- integer, parameter, public mem_usage_i = 55

  file number of memory usage file
- integer, parameter, public output_ev_i = 56

  file number of output of EV
- integer, parameter, public decomp_i = 57

  file number of output of EV decomposition
- integer, parameter, public hel_export_i = 59

  file number of output of HELENA equilibrium export file
- integer, parameter, public hel_pert_i = 58

  file number of HELENA equilibrium perturbation file
- integer, parameter, public prop_b_tor_i = 60

  file number of B_ϕ proportionality factor file

B.27.1 Detailed Description

Numerical variables used by most other modules.

B.27.2 Variable Documentation

B.27.2.1 alpha_style

```
integer, public num_vars::alpha_style
```

style for alpha (1: one field line, many turns, 2: many field lines, one turn)

Definition at line 100 of file num_vars.f90.

B.27.2.2 bc_style

```
integer, dimension(2), public num_vars::bc_style
```

style for BC left and right

Definition at line 94 of file num_vars.f90.

B.27.2.3 compare_tor_pos

```
logical, public num_vars::compare_tor_pos = .false.
```

compare quantities at toroidal positions (only for POST)

Definition at line 151 of file num_vars.f90.

B.27.2.4 data_dir

```
character(len=4), public num_vars::data_dir = 'Data'
```

directory where to save data for plots

Definition at line 155 of file num_vars.f90.

B.27.2.5 debug_version

```
logical public num_vars::debug_version = .true.
```

debug version used

not debug version

Definition at line 62 of file num_vars.f90.

B.27.2.6 decomp_i

```
integer, parameter, public num_vars::decomp_i = 57
```

file number of output of EV decomposition

Definition at line 189 of file num_vars.f90.

B.27.2.7 do_execute_command_line

```
logical, public num_vars::do_execute_command_line = .false.
```

call ”execute_command_line” inside program

Definition at line 148 of file num_vars.f90.

B.27.2.8 dp

```
integer, parameter, public num_vars::dp = REAL64
```

double precision

Definition at line 46 of file num_vars.f90.

B.27.2.9 dpi

```
integer, parameter, public num_vars::dpi = INT64
```

double precision

Definition at line 48 of file num_vars.f90.

B.27.2.10 eq_i

```
integer, parameter, public num_vars::eq_i = 51
```

file number of equilibrium file from VMEC or HELENA

Definition at line 183 of file num_vars.f90.

B.27.2.11 eq_job_nr

```
integer, public num_vars::eq_job_nr
```

nr. of eq job

Definition at line 79 of file num_vars.f90.

B.27.2.12 eq_jobs_lims

```
integer, dimension(:,:), allocatable, public num_vars::eq_jobs_lims
```

data about eq jobs: [$\min_\theta$, $\max_\theta$] for all jobs

Definition at line 77 of file num_vars.f90.

B.27.2.13 eq_name

```
character(len=max_str_ln), public num_vars::eq_name
```

name of equilibrium file from VMEC or HELENA

Definition at line 138 of file num_vars.f90.

B.27.2.14 eq_style

```
integer, public num_vars::eq_style
```

either 1 (VMEC) or 2 (HELENA)

Definition at line 89 of file num_vars.f90.

B.27.2.15 ev_bc

```
real(dp), public num_vars::ev_bc
```

value of artificial Eigenvalue for boundary condition

Definition at line 116 of file num_vars.f90.

B.27.2.16 ev_guess

```
real(dp), public num_vars::ev_guess
```

first guess for eigenvalue

Definition at line 117 of file num_vars.f90.

B.27.2.17 ev_style

```
integer, public num_vars::ev_style
```

determines the method used for solving an EV problem

Definition at line 88 of file num_vars.f90.

B.27.2.18 ex_max_size

```
integer, parameter, public num_vars::ex_max_size = 10000
```

maximum size of matrices for external plot

Definition at line 137 of file num_vars.f90.

B.27.2.19 ex_plot_style

```
integer, public num_vars::ex_plot_style
```

external plot style (1: GNUPlot, 2: Bokeh for 2D, Mayavi for 3D)

Definition at line 175 of file num_vars.f90.

B.27.2.20 export_hel

```
logical, public num_vars::export_hel = .false.
```

export HELENA

Definition at line 142 of file num_vars.f90.

B.27.2.21 hel_export_i

```
integer, parameter, public num_vars::hel_export_i = 59
```

file number of output of HELENA equilibrium export file

Definition at line 190 of file num_vars.f90.

B.27.2.22 hel_pert_i

```
integer, parameter, public num_vars::hel_pert_i = 58
```

file number of HELENA equilibrium perturbation file

Definition at line 191 of file num_vars.f90.

B.27.2.23 input_i

```
integer, parameter, public num_vars::input_i = 50
```

file number of input file

Definition at line 182 of file num_vars.f90.

B.27.2.24 input_name

```
character(len=max_str_ln), public num_vars::input_name
```

will hold the full name of the input file

Definition at line 174 of file num_vars.f90.

B.27.2.25 invert_top_bottom_h

```
logical, public num_vars::invert_top_bottom_h = .false.
```

invert top and bottom for HELENA equilibria

Definition at line 144 of file num_vars.f90.

B.27.2.26 iu

```
complex(dp), parameter, public num_vars::iu = (0, 1)
```

complex unit

Definition at line 85 of file num_vars.f90.

B.27.2.27 jump_to_sol

```
logical, public num_vars::jump_to_sol = .false.
```

jump to solution

Definition at line 141 of file num_vars.f90.

B.27.2.28 k_style

```
integer, public num_vars::k_style
```

style for kinetic energy

Definition at line 93 of file num_vars.f90.

B.27.2.29 ltest

```
logical, public num_vars::ltest
```

whether or not to call the testing routines

Definition at line 112 of file num_vars.f90.

B.27.2.30 magn_int_style

```
integer, public num_vars::magn_int_style
```

style for magnetic integrals (1: trapezoidal, 2: Simpson 3/8)

Definition at line 124 of file num_vars.f90.

B.27.2.31 matrix_slepc_style

```
integer, public num_vars::matrix_slepc_style
```

style for matrix storage (1: sparse, 2: shell)

Definition at line 96 of file num_vars.f90.

B.27.2.32 max_deriv

```
integer, parameter, public num_vars::max_deriv = 2
```

highest derivatives for metric factors in Flux coords.

Definition at line 52 of file num_vars.f90.

B.27.2.33 max_it_rich

```
integer, public num_vars::max_it_rich
```

number of levels for Richardson extrapolation

Definition at line 127 of file num_vars.f90.

B.27.2.34 max_it_slepc

```
integer, public num_vars::max_it_slepc
```

maximum nr. of iterations for SLEPC

Definition at line 101 of file num_vars.f90.

B.27.2.35 max_it_zero

```
integer, public num_vars::max_it_zero
```

maximum number of iterations to find zeros

Definition at line 131 of file num_vars.f90.

B.27.2.36 max_name_ln

```
integer, parameter, public num_vars::max_name_ln = 30
```

maximum length of filenames

Definition at line 51 of file num_vars.f90.

B.27.2.37 max_njq_change

```
real(dp), public num_vars::max_njq_change
```

maximum change of prim. mode number times saf. fac. / rot. transf. when using X_style 2 (fast)

Definition at line 119 of file num_vars.f90.

B.27.2.38 max_nr_backtracks_hh

```
integer, public num_vars::max_nr_backtracks_hh
```

maximum number of backtracks for Householder, relax. factors

Definition at line 132 of file num_vars.f90.

B.27.2.39 max_r_plot

```
real(dp), public num_vars::max_r_plot
```

max. of `r_plot`

Definition at line 164 of file num_vars.f90.

B.27.2.40 max_rvac_plot

```
real(dp), public num_vars::max_rvac_plot
```

max. of `R` in which to plot vacuum

Definition at line 166 of file num_vars.f90.

B.27.2.41 max_str_ln

```
integer, parameter, public num_vars::max_str_ln = 120
```

maximum length of strings

Definition at line 50 of file num_vars.f90.

B.27.2.42 max_theta_plot

```
real(dp), public num_vars::max_theta_plot
```

max. of `theta_plot`

Definition at line 160 of file num_vars.f90.

B.27.2.43 max_tot_mem

```
real(dp), public num_vars::max_tot_mem
```

maximum total memory for all processes [MB]

Definition at line 74 of file num_vars.f90.

B.27.2.44 max_x_mem

```
real(dp), public num_vars::max_x_mem
```

maximum memory for perturbation calculations for all processes [MB]

Definition at line 75 of file num_vars.f90.

B.27.2.45 max_zeta_plot

```
real(dp), public num_vars::max_zeta_plot
```

max. of `zeta_plot`

Definition at line 162 of file num_vars.f90.

B.27.2.46 max_zvac_plot

```
real(dp), public num_vars::max_zvac_plot
```

max. of `Z` in which to plot vacuum

Definition at line 168 of file num_vars.f90.

B.27.2.47 mem_scale_fac

```
real(dp), parameter, public num_vars::mem_scale_fac = 6.0
```

empirical scale factor of memory to calculate eq compared to just storing it

Definition at line 80 of file num_vars.f90.

B.27.2.48 mem_usage_count

```
integer, public num_vars::mem_usage_count
```

counter for memory usage output

Definition at line 58 of file num_vars.f90.

B.27.2.49 mem_usage_i

```
integer, parameter, public num_vars::mem_usage_i = 55
```

file number of memory usage file

Definition at line 187 of file num_vars.f90.

B.27.2.50 mem_usage_name

```
character(len=9), parameter, public num_vars::mem_usage_name = 'mem_usage'
```

name of memory usage file

Definition at line 57 of file num_vars.f90.

B.27.2.51 min_pb3d_version

```
real(dp), parameter, public num_vars::min_pb3d_version = 2.43_dp
```

minimum PB3D version for POST

Definition at line 60 of file num_vars.f90.

B.27.2.52 min_r_plot

```
real(dp), public num_vars::min_r_plot
```

min. of `r_plot`

Definition at line 163 of file num_vars.f90.

B.27.2.53 min_rvac_plot

```
real(dp), public num_vars::min_rvac_plot
```

min. of `R` in which to plot vacuum

Definition at line 165 of file num_vars.f90.

B.27.2.54 min_theta_plot

```
real(dp), public num_vars::min_theta_plot
```

min. of `theta_plot`

Definition at line 159 of file num_vars.f90.

B.27.2.55 min_zeta_plot

```
real(dp), public num_vars::min_zeta_plot
```

min. of `zeta_plot`

Definition at line 161 of file num_vars.f90.

B.27.2.56 min_zvac_plot

```
real(dp), public num_vars::min_zvac_plot
```

min. of Z in which to plot vacuum

Definition at line 167 of file num_vars.f90.

B.27.2.57 mu_0_original

```
real(dp), parameter, public num_vars::mu_0_original = 4E-7_dp*pi
```

permeability of free space

Definition at line 84 of file num_vars.f90.

B.27.2.58 n_procs

```
integer, public num_vars::n_procs
```

nr. of MPI processes

Definition at line 69 of file num_vars.f90.

B.27.2.59 n_sol_plotted

```
integer, dimension(4), public num_vars::n_sol_plotted
```

how many solutions to be plot (first unstable, last unstable, first stable, last stable)

Definition at line 171 of file num_vars.f90.

B.27.2.60 n_sol_requested

```
integer, public num_vars::n_sol_requested
```

how many solutions requested

Definition at line 170 of file num_vars.f90.

B.27.2.61 n_theta_plot

```
integer, public num_vars::n_theta_plot
```

nr. of poloidal points in plot

Definition at line 156 of file num_vars.f90.

B.27.2.62 n_vac_plot

```
integer, dimension(2), public num_vars::n_vac_plot
```

nr. of points in R and Z in vacuum

Definition at line 158 of file num_vars.f90.

B.27.2.63 n_zeta_plot

```
integer, public num_vars::n_zeta_plot
```

nr. of toroidal points in plot

Definition at line 157 of file num_vars.f90.

B.27.2.64 no_output

```
logical, public num_vars::no_output = .false.
```

no output shown

Definition at line 145 of file num_vars.f90.

B.27.2.65 no_plots

```
logical, public num_vars::no_plots = .false.
```

no plots made

Definition at line 140 of file num_vars.f90.

B.27.2.66 norm_disc_prec_eq

```
integer, public num_vars::norm_disc_prec_eq
```

precision for normal discretization for equilibrium

Definition at line 120 of file num_vars.f90.

B.27.2.67 norm_disc_prec_sol

```
integer, public num_vars::norm_disc_prec_sol
```

precision for normal discretization for solution

Definition at line 122 of file num_vars.f90.

B.27.2.68 norm_disc_prec_x

```
integer, public num_vars::norm_disc_prec_x
```

precision for normal discretization for perturbation

Definition at line 121 of file num_vars.f90.

B.27.2.69 norm_disc_style_sol

```
integer, public num_vars::norm_disc_style_sol
```

style for normal discretization for solution (1: central fin. diff., 2: left fin. diff.)

Definition at line 123 of file num_vars.f90.

B.27.2.70 norm_style

```
integer, public num_vars::norm_style
```

style for normalization

Definition at line 92 of file num_vars.f90.

B.27.2.71 output_ev_i

```
integer, parameter, public num_vars::output_ev_i = 56
```

file number of output of EV

Definition at line 188 of file num_vars.f90.

B.27.2.72 output_i

```
integer, parameter, public num_vars::output_i = 53
```

file number of output file

Definition at line 185 of file num_vars.f90.

B.27.2.73 output_name

```
character(len=3), parameter, public num_vars::output_name = 'out'
```

name of output file

Definition at line 55 of file num_vars.f90.

B.27.2.74 pb3d_i

```
integer, parameter, public num_vars::pb3d_i = 52
```

file number of PB3D output file

Definition at line 184 of file num_vars.f90.

B.27.2.75 pb3d_name

```
character(len=max_str_ln), public num_vars::pb3d_name
```

name of PB3D output file

Definition at line 139 of file num_vars.f90.

B.27.2.76 pert_mult_factor_post

```
real(dp), public num_vars::pert_mult_factor_post
```

factor with which to multiply perturbation strength for POST

Definition at line 176 of file num_vars.f90.

B.27.2.77 pi

```
real(dp), parameter, public num_vars::pi =4_dp*datan(1.0_dp)
```

π

Definition at line 83 of file num_vars.f90.

B.27.2.78 plot_b

```
logical, public num_vars::plot_b
```

whether to plot the magnetic field in real coordinates

Definition at line 104 of file num_vars.f90.

B.27.2.79 plot_dir

```
character(len=5), public num_vars::plot_dir = 'Plots'
```

directory where to save plots

Definition at line 153 of file num_vars.f90.

B.27.2.80 plot_e_rec

```
logical, public num_vars::plot_e_rec
```

whether to plot energy reconstruction in POST

Definition at line 111 of file num_vars.f90.

B.27.2.81 plot_flux_q

```
logical, public num_vars::plot_flux_q
```

whether to plot flux quantities in real coordinates

Definition at line 106 of file num_vars.f90.

B.27.2.82 plot_grid_style

```
integer, public num_vars::plot_grid_style
```

style for POST plot grid (0: 3-D plots, 1: slab plots, 2: slab plots with folding, 3: straight cylinder))

Definition at line 169 of file num_vars.f90.

B.27.2.83 plot_j

```
logical, public num_vars::plot_j
```

whether to plot the current in real coordinates

Definition at line 105 of file num_vars.f90.

B.27.2.84 plot_kappa

```
logical, public num_vars::plot_kappa
```

whether to plot curvature

Definition at line 107 of file num_vars.f90.

B.27.2.85 plot_magn_grid

```
logical, public num_vars::plot_magn_grid
```

whether to plot the grid in real coordinates

Definition at line 103 of file num_vars.f90.

B.27.2.86 plot_resonance

```
logical, public num_vars::plot_resonance
```

whether to plot the q-profile or iota-profile with resonances

Definition at line 102 of file num_vars.f90.

B.27.2.87 plot_size

```
integer, dimension(2), public num_vars::plot_size
```

size of plot in inches

Definition at line 172 of file num_vars.f90.

B.27.2.88 plot_sol_q

```
logical, public num_vars::plot_sol_q
```

whether to plot magnetic perturbation of solution in POST

Definition at line 109 of file num_vars.f90.

B.27.2.89 plot_sol_xi

```
logical, public num_vars::plot_sol_xi
```

whether to plot plasma perturbation of solution in POST

Definition at line 108 of file num_vars.f90.

B.27.2.90 plot_vac_pot

```
logical, public num_vars::plot_vac_pot
```

whether to plot vacuum potential in POST

Definition at line 110 of file num_vars.f90.

B.27.2.91 plot_vmec_modes

```
logical, public num_vars::plot_vmec_modes = .false.
```

plot VMEC modes

Definition at line 143 of file num_vars.f90.

B.27.2.92 post_output_full

```
logical, public num_vars::post_output_full = .false.
```

POST has output on full grids.

Definition at line 146 of file num_vars.f90.

B.27.2.93 post_output_sol

```
logical, public num_vars::post_output_sol = .false.
```

POST has outputs of solution.

Definition at line 147 of file num_vars.f90.

B.27.2.94 post_style

```
integer, public num_vars::post_style
```

style for POST (1: extended grid, 2: B-aligned grid)

Definition at line 98 of file num_vars.f90.

B.27.2.95 print_mem_usage

```
logical, public num_vars::print_mem_usage = .false.
```

print memory usage is printed

Definition at line 149 of file num_vars.f90.

B.27.2.96 prog_name

```
character(len=4), public num_vars::prog_name
```

name of program, used for info

Definition at line 54 of file num_vars.f90.

B.27.2.97 prog_style

```
integer, public num_vars::prog_style
```

program style (1: PB3D, 2: PB3D_POST)

Definition at line 53 of file num_vars.f90.

B.27.2.98 prog_version

```
real(dp), parameter, public num_vars::prog_version = 2.47_dp
```

version number

Definition at line 59 of file num_vars.f90.

B.27.2.99 prop_b_tor_i

```
integer, parameter, public num_vars::prop_b_tor_i = 60
```

file number of B_ϕ proportionality factor file

Definition at line 192 of file num_vars.f90.

B.27.2.100 rank

```
integer, public num_vars::rank
```

MPI rank.

Definition at line 68 of file num_vars.f90.

B.27.2.101 retain_all_sol

```
logical, public num_vars::retain_all_sol
```

retain also faulty solutions

Definition at line 152 of file num_vars.f90.

B.27.2.102 rho_style

```
integer, public num_vars::rho_style
```

style for equilibrium density profile

Definition at line 90 of file num_vars.f90.

B.27.2.103 rich_restart_lvl

```
integer, public num_vars::rich_restart_lvl
```

starting Richardson level (0: none [default])

Definition at line 173 of file num_vars.f90.

B.27.2.104 rz_0

```
real(dp), dimension(2), public num_vars::rz_0
```

origin of geometrical poloidal coordinate

Definition at line 179 of file num_vars.f90.

B.27.2.105 script_dir

```
character(len=7), public num_vars::script_dir = 'Scripts'
```

directory where to save scripts for plots

Definition at line 154 of file num_vars.f90.

B.27.2.106 shell_commands_i

```
integer, parameter, public num_vars::shell_commands_i = 54
```

file number of shell commands file

Definition at line 186 of file num_vars.f90.

B.27.2.107 shell_commands_name

```
character(len=14), parameter, public num_vars::shell_commands_name = 'shell_commands'
```

name of shell commands file

Definition at line 56 of file num_vars.f90.

B.27.2.108 sol_n_procs

```
integer, public num_vars::sol_n_procs
```

nr. of MPI processes for solution with SLEPC

Definition at line 70 of file num_vars.f90.

B.27.2.109 solver_slepc_style

```
integer, public num_vars::solver_slepc_style
```

style for solver (1: Krylov-Schur, 2: GD)

Definition at line 97 of file num_vars.f90.

B.27.2.110 swap_angles

```
logical, public num_vars::swap_angles = .false.
```

swap angles theta and zeta in plots (only for POST)

Definition at line 150 of file num_vars.f90.

B.27.2.111 time_start

```
integer(kind=8), public num_vars::time_start
```

start time of simulation

Definition at line 71 of file num_vars.f90.

B.27.2.112 tol_norm

```
real(dp), public num_vars::tol_norm
```

tolerance for normal range (normalized to 0..1)

Definition at line 134 of file num_vars.f90.

B.27.2.113 tol_rich

```
real(dp), public num_vars::tol_rich
```

tolerance for Richardson extrapolation

Definition at line 128 of file num_vars.f90.

B.27.2.114 tol_slepc

```
real(dp), dimension(:), allocatable, public num_vars::tol_slepc
```

tolerance for SLEPC for different Richardson levels

Definition at line 118 of file num_vars.f90.

B.27.2.115 tol_zero

```
real(dp), public num_vars::tol_zero
```

tolerance for zeros

Definition at line 133 of file num_vars.f90.

B.27.2.116 u_style

```
integer, public num_vars::u_style
```

style for calculation of U (1: ord.2, 2: ord.1, 1: ord.0)

Definition at line 91 of file num_vars.f90.

B.27.2.117 use_normalization

```
logical, public num_vars::use_normalization
```

whether to use normalization or not

Definition at line 115 of file num_vars.f90.

B.27.2.118 use_pol_flux_e

```
logical, public num_vars::use_pol_flux_e
```

whether poloidal flux is used in E coords.

Definition at line 113 of file num_vars.f90.

B.27.2.119 use_pol_flux_f

```
logical, public num_vars::use_pol_flux_f
```

whether poloidal flux is used in F coords.

Definition at line 114 of file num_vars.f90.

B.27.2.120 weight_dp

```
real(dp), parameter, public num_vars::weight_dp = 0.008
```

size of double precision in kB

Definition at line 49 of file num_vars.f90.

B.27.2.121 x_grid_style

```
integer, public num_vars::x_grid_style
```

style for normal component of X grid (1: eq, 2: sol, 3: enriched)

Definition at line 99 of file num_vars.f90.

B.27.2.122 x_job_nr

```
integer, public num_vars::x_job_nr
```

nr. of X job

Definition at line 78 of file num_vars.f90.

B.27.2.123 x_jobs_lims

```
integer, dimension(:,:), allocatable, public num_vars::x_jobs_lims
```

data about X jobs: [$\min_k$, $\max_k$, $\min_m$, $\max_m$] for all jobs

Definition at line 76 of file num_vars.f90.

B.27.2.124 x_style

```
integer, public num_vars::x_style
```

style for secondary mode numbers (1: prescribed, 2: fast)

Definition at line 95 of file num_vars.f90.

B.28 output_ops Module Reference

Operations concerning giving output, on the screen as well as in output files.

Interfaces and Types

- interface print_ex_2d

  Print 2-D output on a file.
- interface print_ex_3d

  Print 3-D output on a file.
- interface plot_hdf5

  Prints variables `vars` with names `var_names` in an HDF5 file with name c file_name and accompanying XDMF file.

Functions/Subroutines

- subroutine, public draw_ex (var_names, draw_name, nplt, draw_dim, plot_on_screen, ex_plot_style, data_name, draw_ops, extra_ops, is_animated, ranges, delay, persistent)

  Use external program to draw a plot.
- subroutine, public plot_diff_hdf5 (a, b, file_name, tot_dim, loc_offset, descr, output_message)

  Takes two input vectors and plots these as well as the relative and absolute difference in a HDF5 file.

B.28.1 Detailed Description

Operations concerning giving output, on the screen as well as in output files.

B.28.2 Function/Subroutine Documentation

B.28.2.1 draw_ex()

```
subroutine, public output_ops::draw_ex (
                character(len=*), dimension(:), intent(in) var_names,
                character(len=*), intent(in) draw_name,
                integer, intent(in) nplt,
                integer, intent(in) draw_dim,
                logical, intent(in) plot_on_screen,
                integer, intent(in), optional ex_plot_style,
                character(len=*), intent(in), optional data_name,
                character(len=*), dimension(:), intent(in), optional draw_ops,
                character(len=*), intent(in), optional extra_ops,
                logical, intent(in), optional is_animated,
                real(dp), dimension(:,:), intent(in), optional ranges,
                integer, intent(in), optional delay,
                logical, intent(in), optional persistent)
```

Use external program to draw a plot.

The external programs used are determined by `ex_plot_style`:

- `ex_plot_style` = 1: Use GnuPlot for 2-D and 3-D
- `ex_plot_style` = 1: Use Bokeh in 2-D and Mayavi in 3-D

The output is saved in a file `[draw_name].[ext]` where the extension depends on the plotting style and the dimension, or on the screen, depending on `plot_on_screen`.

If not specified otherwise through data_name, it is assumed that the datafile to be plot is situated in `[draw_name].dat`.

The title(s) of the plot(s) are provided through `var_names` and the number of plots through `nplt`. If there are less names than the number of plots, the first one is taken and appended by the plot number.

Furthermore, `draw_dim` determines whether the plot is to be 2-D, 3-D or 2-D slices in 3-D.

Also, an optional command `extra_ops` can be provided, to provide overal options, as well as draw_ops that specifies the line style for the plots from the file. If `less` draw_ops are provided than plots, they will be cycled.

Finally, there is an option to provide animated plots, but is only available in 2-D. Also, for external plot style 1, no on-screen view is possible. Animations can take optional ranges arguments as well as delay.

Note

About draw_dim:

- `draw_dim` = 1: 2-D plot; should be called with the output of output_ops.print_ex_2d(), `nplt` should be correctly set.
- `draw_dim` = 2: 3-D plot; should be called with the output of print_ex_3D. For GNUPlot, `nplt` is ignored, but not for Mayavi, as it needs this information to be able to reconstruct 2D arrays for X, Y and Z.
- `draw_dim` = 3: 2-D plot in 3-D slices; should be called with the output of output_ops.print_ex_2d(), `nplt` should be correctly set.

Parameters

in	*var_names*	name of variables

in	*draw_name*	name of drawing
in	*nplt*	number of plots
in	*draw_dim*	1: 2-D, 2: 3-D, 3: decoupled 3-D
in	*plot_on_screen*	True if on screen, false if in file
in	*ex_plot_style*	alternative external plot style
in	*data_name*	name of data file
in	*draw_ops*	drawing options
in	*extra_ops*	extra options
in	*is_animated*	plot is animated
in	*ranges*	x and y range of animated plot
in	*delay*	time delay between animated plot frames
in	*persistent*	keep on-screen plot open

Definition at line 1076 of file output_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.28.2.2 plot_diff_hdf5()

```
subroutine, public output_ops::plot_diff_hdf5 (
            real(dp), dimension(:,:,:), intent(in) a,
            real(dp), dimension(:,:,:), intent(in) b,
            character(len=*), intent(in) file_name,
            integer, dimension(3), intent(in), optional tot_dim,
            integer, dimension(3), intent(in), optional loc_offset,
            character(len=*), intent(in), optional descr,
            logical, intent(in), optional output_message)
```

Takes two input vectors and plots these as well as the relative and absolute difference in a HDF5 file.

This is similar to a basic version of output_ops.plot_hdf5().

Optionally, an output message can be displayed on screen with the maximum relative and absolute error.

Parameters

in	*a*	vector A
in	*b*	vector B

in	*file_name*	name of plot
in	*tot_dim*	total dimensions of the arrays
in	*loc_offset*	offset of local dimensions
in	*descr*	description
in	*output_message*	whether to display a message or not

Definition at line 1763 of file output_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.29 pb3d_ops Module Reference

Operations on PB3D output.

Functions/Subroutines

- integer function, public reconstruct_pb3d_in (data_name)

  Reconstructs the input variables from PB3D HDF5 output.
- integer function, public reconstruct_pb3d_grid (grid, data_name, rich_lvl, tot_rich, lim_pos, grid_limits)

  Reconstructs grid variables from PB3D HDF5 output.
- integer function, public reconstruct_pb3d_eq_1 (grid_eq, eq, data_name, lim_pos)

  Reconstructs the equilibrium variables from PB3D HDF5 output.

- integer function, public reconstruct_pb3d_eq_2 (grid_eq, eq, data_name, rich_lvl, tot_rich, lim_pos)

  Reconstructs the equilibrium variables from PB3D HDF5 output.
- integer function, public reconstruct_pb3d_x_1 (mds, grid_x, x, data_name, rich_lvl, tot_rich, lim_sec_x, lim_pos)

  Reconstructs the vectorial perturbation variables from PB3D HDF5 output.
- integer function, public reconstruct_pb3d_x_2 (mds, grid_x, x, data_name, rich_lvl, tot_rich, lim_sec_x, lim_pos, is_field_averaged)

  Reconstructs the tensorial perturbation variables from PB3D HDF5 output.
- integer function, public reconstruct_pb3d_vac (vac, data_name, rich_lvl)

  Reconstructs the vacuum variables from PB3D HDF5 output.
- integer function, public reconstruct_pb3d_sol (mds, grid_sol, sol, data_name, rich_lvl, lim_sec_sol, lim_pos)

  Reconstructs the solution variables from PB3D HDF5 output.
- integer function, public get_pb3d_grid_size (n, grid_name, rich_lvl, tot_rich)

  get grid size

B.29.1 Detailed Description

Operations on PB3D output.

Note

If you have parallel jobs, if the reconstruction routines are called for multiple equilibrium jobs, this can only be done after the last equilibrium job is finished. There are no checks for this.

B.29.2 Function/Subroutine Documentation

B.29.2.1 get_pb3d_grid_size()

```
integer function, public pb3d_ops::get_pb3d_grid_size (
                integer, dimension(3), intent(inout) n,
                character(len=*), intent(in) grid_name,
                integer, intent(in), optional rich_lvl,
                logical, intent(in), optional tot_rich)
```

get grid size

Note

`grid_` is added in front the grid_name.

Returns

ierr

Parameters

in,out	*n*	n of grid

in	*grid_name*	name of grid
in	*rich_lvl*	Richardson level to reconstruct
in	*tot_rich*	whether to combine with previous Richardson levels

Definition at line 1452 of file PB3D_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.29.2.2 reconstruct_pb3d_eq_1()

```
integer function, public pb3d_ops::reconstruct_pb3d_eq_1 (
            type(grid_type), intent(in) grid_eq,
            type(eq_1_type), intent(inout), optional eq,
            character(len=*), intent(in) data_name,
            integer, dimension(1,2), intent(in), optional lim_pos)
```

Reconstructs the equilibrium variables from PB3D HDF5 output.

Furthermore, using `lim_pos`, you can obtain a subset of the data by directly passing its limits to the underlying HDF5 routine. This refers to the position dimensions only. If provided, the normal limits of a divided grid refer to the subset, as in copy_grid().

Returns

ierr

Parameters

`in`	*grid_eq*	equilibrium grid

in,out	*eq*	flux equilibrium
in	*data_name*	name to reconstruct
in	*lim_pos*	position limits of subset of data

Definition at line 595 of file PB3D_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.29.2.3 reconstruct_pb3d_eq_2()

```
integer function, public pb3d_ops::reconstruct_pb3d_eq_2 (
                type(grid_type), intent(in) grid_eq,
                type(eq_2_type), intent(inout) eq,
                character(len=*), intent(in) data_name,
                integer, intent(in), optional rich_lvl,
                logical, intent(in), optional tot_rich,
                integer, dimension(3,2), intent(in), optional lim_pos)
```

Reconstructs the equilibrium variables from PB3D HDF5 output.

Also, if `rich_lvl` is provided, *R*[`rich_lvl`] is appended to the data name if it is >0.

With `tot_rich` the information from previous Richardson levels can be combined.

Furthermore, using `lim_pos`, you can obtain a subset of the data by directly passing its limits to the underlying HDF5 routine. This refers to the position dimensions only. If provided, the normal limits of a divided grid refer to the subset, as in copy_grid().

Returns

ierr

Parameters

`in`	*grid_eq*	equilibrium grid

in,out	*eq*	metric equilibrium
in	*data_name*	name to reconstruct
in	*rich_lvl*	Richardson level to reconstruct
in	*tot_rich*	whether to combine with previous Richardson levels
in	*lim_pos*	position limits of subset of data

Definition at line 693 of file PB3D_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.29.2.4 reconstruct_pb3d_grid()

```
integer function, public pb3d_ops::reconstruct_pb3d_grid (
            type(grid_type), intent(inout) grid,
            character(len=*), intent(in) data_name,
            integer, intent(in), optional rich_lvl,
            logical, intent(in), optional tot_rich,
            integer, dimension(3,2), intent(in), optional lim_pos,
            integer, dimension(2), intent(in), optional grid_limits)
```

Reconstructs grid variables from PB3D HDF5 output.

Also, if `rich_lvl` is provided, *R*[`rich_lvl`] is appended to the data name if it is >0.

With `tot_rich` the information from previous Richardson levels can be combined.

Furthermore, using `lim_pos`, you can obtain a subset of the data by directly passing its limits to the underlying HDF5 routine. This refers to the position dimensions only. If provided, the normal limits of a divided grid refer to the subset, as in copy_grid().

Note

1. `grid_` is added in front the data_name.
2. By providing `lim_pos` equal to 0 in the angular dimensions, the angular part can be discarded when reconstructing the grid.

Returns

ierr

Parameters

in	*data_name*	name of grid
in	*rich_lvl*	Richardson level to reconstruct
in	*tot_rich*	whether to combine with previous Richardson levels
in	*lim_pos*	position limits of subset of data
in	*grid_limits*	i_limit of grid

Definition at line 440 of file PB3D_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.29.2.5 reconstruct_pb3d_in()

```
integer function, public pb3d_ops::reconstruct_pb3d_in (
            character(len=*), intent(in) data_name)
```

Reconstructs the input variables from PB3D HDF5 output.

Returns

ierr

Parameters

in	*data_name*	name of input variables

Definition at line 40 of file PB3D_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.29.2.6 reconstruct_pb3d_sol()

```
integer function, public pb3d_ops::reconstruct_pb3d_sol (
                type(modes_type), intent(in) mds,
                type(grid_type), intent(in) grid_sol,
                type(sol_type), intent(inout) sol,
                character(len=*), intent(in) data_name,
                integer, intent(in), optional rich_lvl,
                integer, dimension(2), intent(in), optional lim_sec_sol,
                integer, dimension(1,2), intent(in), optional lim_pos)
```

Reconstructs the solution variables from PB3D HDF5 output.

Also, if `rich_lvl` is provided, *R*[`rich_lvl`] is appended to the data name if it is >0.

Furthermore, using `lim_pos`, you can obtain a subset of the data by directly passing its limits to the underlying HDF5 routine. This refers to the position dimensions only. If provided, the normal limits of a divided grid refer to the subset, as in copy_grid().

`lim_sec_sol`, on the other hand, selects a range of mode numbers.

Returns

ierr

Parameters

in	*mds*	general modes variables
in	*grid_sol*	solution grid
in,out	*sol*	solution variables
in	*data_name*	name to reconstruct
in	*rich_lvl*	Richardson level to reconstruct
in	*lim_sec_sol*	limits of m_X (pol. flux) or n_X (tor. flux)
in	*lim_pos*	position limits of subset of data

Definition at line 1357 of file PB3D_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.29.2.7 reconstruct_pb3d_vac()

```
integer function, public pb3d_ops::reconstruct_pb3d_vac (
            type(vac_type), intent(inout) vac,
            character(len=*), intent(in) data_name,
            integer, intent(in), optional rich_lvl)
```

Reconstructs the vacuum variables from PB3D HDF5 output.

Also, if `rich_lvl` is provided, R[`rich_lvl`] is appended to the data name if it is >0.

Returns

ierr

Parameters

in,out	*vac*	vacuum variables
in	*data_name*	name to reconstruct
in	*rich_lvl*	Richardson level to reconstruct

Definition at line 1227 of file PB3D_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.29.2.8 reconstruct_pb3d_x_1()

```
integer function, public pb3d_ops::reconstruct_pb3d_x_1 (
            type(modes_type), intent(in) mds,
            type(grid_type), intent(in) grid_x,
            type(x_1_type), intent(inout) x,
            character(len=*), intent(in) data_name,
            integer, intent(in), optional rich_lvl,
            logical, intent(in), optional tot_rich,
            integer, dimension(2), intent(in), optional lim_sec_x,
            integer, dimension(3,2), intent(in), optional lim_pos)
```

Reconstructs the vectorial perturbation variables from PB3D HDF5 output.

Also, if `rich_lvl` is provided, *R*[`rich_lvl`] is appended to the data name if it is >0.

With `tot_rich` the information from previous Richardson levels can be combined.

Furthermore, using `lim_pos`, you can obtain a subset of the data by directly passing its limits to the underlying HDF5 routine. This refers to the position dimensions only. If provided, the normal limits of a divided grid refer to the subset, as in copy_grid().

`lim_sec_X`, on the other hand, selects a range of mode numbers.

Returns

ierr

Parameters

in	*mds*	general modes variables
in	*grid_x*	perturbation grid
in,out	*x*	vectorial perturbation variables
in	*data_name*	name to reconstruct
in	*rich_lvl*	Richardson level to reconstruct
in	*tot_rich*	whether to combine with previous Richardson levels

in	*lim_sec_x*	limits of m_X (pol. flux) or n_X (tor. flux)
in	*lim_pos*	position limits of subset of data

Definition at line 831 of file PB3D_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.29.2.9 reconstruct_pb3d_x_2()

```
integer function, public pb3d_ops::reconstruct_pb3d_x_2 (
                type(modes_type), intent(in) mds,
                type(grid_type), intent(in) grid_x,
                type(x_2_type), intent(inout) x,
                character(len=*), intent(in) data_name,
                integer, intent(in), optional rich_lvl,
                logical, intent(in), optional tot_rich,
                integer, dimension(2,2), intent(in), optional lim_sec_x,
                integer, dimension(3,2), intent(in), optional lim_pos,
                logical, intent(in), optional is_field_averaged)
```

Reconstructs the tensorial perturbation variables from PB3D HDF5 output.

Also, if `rich_lvl` is provided, R[`rich_lvl`] is appended to the data name if it is >0.

With `tot_rich` the information from previous Richardson levels can be combined.

Furthermore, using `lim_pos`, you can obtain a subset of the data by directly passing its limits to the underlying HDF5 routine. This refers to the position dimensions only. If provided, the normal limits of a divided grid refer to the subset, as in copy_grid().

`lim_sec_X`, on the other hand, selects a range of mode numbers.

Note

The tensorial perturbation type can also be used for field- aligned variables, in which case the first index is assumed to have dimension 1 only. This can be triggered using `is_field_averaged`.

Returns

ierr

Parameters

in	*mds*	general modes variables
in	*grid_x*	perturbation grid
in,out	*x*	tensorial perturbation vars
in	*data_name*	name to reconstruct
in	*rich_lvl*	Richardson level to reconstruct
in	*tot_rich*	whether to combine with previous Richardson levels
in	*lim_sec_x*	limits of m_X (pol flux) or n_X (tor flux) for both dimensions
in	*lim_pos*	position limits of subset of data
in	*is_field_averaged*	if field-averaged, only one dimension for first index

Definition at line 990 of file PB3D_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.30 pb3d_utilities Module Reference

Numerical utilities related to PB3D operations.

Interfaces and Types

- interface conv_1d2nd

  Converts 1-D to n-D variables.

Functions/Subroutines

- integer function, dimension(3), public setup_par_id (grid, rich_lvl_max, rich_lvl_loc, tot_rich, par_lim, par_id_mem)

  Setup parallel id.
- integer function, dimension(2), public setup_rich_id (rich_lvl_max, tot_rich)

  Returns richardson id.

B.30.1 Detailed Description

Numerical utilities related to PB3D operations.

B.30.2 Function/Subroutine Documentation

B.30.2.1 setup_par_id()

```
integer function, dimension(3), public pb3d_utilities::setup_par_id (
                type(grid_type), intent(in) grid,
                integer, intent(in) rich_lvl_max,
                integer, intent(in) rich_lvl_loc,
                logical, intent(in), optional tot_rich,
                integer, dimension(2), intent(in), optional par_lim,
                integer, dimension(2), intent(inout), optional par_id_mem)
```

Setup parallel id.

The parallel id consists of:

- `par_id(1)`: start index
- `par_id(2)`: end index
- `par_id(3)`: stride

These are set up by considering the transformation between a point $r \in (1 \dots n)$ in the local variable, with corresponding indices $(a \dots b)$ indicated by `par_lim` in the HDF5 variable in memory, so that $n = b - a + 1$:

$$p + ks = a + r - 1,$$

where k is an integer, s is the stride for Richardson level i (with max. level I):

$$\begin{aligned} s &= 2^{I-1} \text{ for } i = 1 \\ &= 2^{I-i+1} \text{ for } i > 1 \end{aligned}$$

and where p is the starting point in the HDF5 variables:

$$\begin{aligned} p &= 1 \text{ for } i = 1 \\ &= 1 + s/2 \text{ for } i > 1 \end{aligned}$$

Therefore, the minimum possible r lies in $(0..s-1)$:

$$\text{mod } (r - 1, s) = r_{\text{min}} - 1,$$

which leads to

$$r_{\text{min}} = 1 + \quad \text{mod } (p - a, s).$$

The maximum possible r, then, lies in $(n - s + 1..n)$, which leads to:

$$r_{\text{max}} = n - s + 1 + \quad \text{mod } (p - b - 1, s).$$

These limits and the strides are saved in `par_id` = $\vec{r} = \begin{pmatrix} r_{\text{min}} \\ r_{\text{max}} \end{pmatrix}$.

If the optional indices a and b are not given they are assumed to be 1 and n, with $n = 1 + ks$, which simplifies the equations to:

$$\begin{aligned} r_{\text{min}} &= p \\ r_{\text{max}} &= n - p + 1. \end{aligned}$$

Optionally, the indices in the HDF5 arrays can also be returned in `par_id_mem` = $\begin{pmatrix} R_{\text{min}} \\ R_{\text{max}} \end{pmatrix}$. These are equal to $k - 1$, where k is the integer refered to above:

$$\vec{R} = 1 + \frac{a - 1 - p + \vec{r}}{s},$$

where the addition between a vector and a scalar should be seen as the element-wise operation.

Parameters

`in`	*rich_lvl_max*	maximum Richardson level
`in`	*rich_lvl_loc*	local Richardson level
`in`	*tot_rich*	whether to combine with previous Richardson levels
`in`	*par_lim*	limits (a..b) of variable requested
`in,out`	*par_id_mem*	parallel id in memory

Returns

parallel id

Definition at line 179 of file PB3D_utilities.f90.

Here is the caller graph for this function:

B.30.2.2 setup_rich_id()

```
integer function, dimension(2), public pb3d_utilities::setup_rich_id (
            integer, intent(in) rich_lvl_max,
            logical, intent(in), optional tot_rich)
```

Returns richardson id.

`rich_id` has the following structure:

- `rich_id(1)`: start Richardson level
- `rich_id(2)`: end Richardson level

Parameters

in	*rich_lvl_max*	maximum Richardson level
in	*tot_rich*	whether to combine with previous Richardson levels

Returns

Richardson id

Definition at line 233 of file PB3D_utilities.f90.

Here is the caller graph for this function:

B.31 rich_ops Module Reference

Operations concerning Richardson extrapolation.

Functions/Subroutines

- integer function, public init_rich ()

  Initialize Richardson extrapolation system.
- subroutine, public term_rich ()

  Terminate the Richardson extrapolation system.
- logical function, public do_rich ()

  Tests whether this Richardson level should be done.
- integer function, public start_rich_lvl ()

  Start a Richardson level.
- subroutine, public stop_rich_lvl ()

  Stop a Richardson level.
- subroutine, public calc_rich_ex (sol_val)

  Calculates the coefficients of the Eigenvalues in the Richardson extrapolation.
- integer function, public find_max_rich_lvl (lvl_rich_req, max_lvl_rich_file)

  Probe to find out which Richardson levels are available.

B.31.1 Detailed Description

Operations concerning Richardson extrapolation.

B.31.2 Function/Subroutine Documentation

B.31.2.1 calc_rich_ex()

```
subroutine, public rich_ops::calc_rich_ex (
            complex(dp), dimension(:), intent(in) sol_val)
```

Calculates the coefficients of the Eigenvalues in the Richardson extrapolation.

This is done using the recursive formula:

$$s(l,i,:) = s(l,i-1,:) + \frac{1}{2^{2\,p\,i}-1}(s(l,i-1,:) - s(l-1,i-1,:)),$$

where s = `sol_val_rich`, i = `ir`, l = `lvl` and p = `norm_disc_prec_sol`, the order of the numerical discretization. as described in [4], algorithm 3.2, p. 306.

Parameters

in	*sol_val*	EV for this Richardson level

Definition at line 458 of file rich_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.31.2.2 do_rich()

```
logical function, public rich_ops::do_rich
```

Tests whether this Richardson level should be done.

Definition at line 312 of file rich_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.31.2.3 find_max_rich_lvl()

```
integer function, public rich_ops::find_max_rich_lvl (
                integer, intent(inout) lvl_rich_req,
                integer, intent(inout) max_lvl_rich_file)
```

Probe to find out which Richardson levels are available.

Returns

ierr

Parameters

in,out	*lvl_rich_req*	requested Richardson level

in,out	*max_lvl_rich_file*	max. Richardson level found in file

Definition at line 537 of file rich_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.31.2.4 init_rich()

```
integer function, public rich_ops::init_rich
```

Initialize Richardson extrapolation system.

Needs to be called only once.

Returns

ierr

Definition at line 27 of file rich_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.31.2.5 start_rich_lvl()

```
integer function, public rich_ops::start_rich_lvl
```

Start a Richardson level.

Calculates `n_par_X` for this level. Then uses this to divide the jobs. equilibrium Finally, the limits are set up for these jobs.

Returns

ierr

Definition at line 326 of file rich_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.31.2.6 stop_rich_lvl()

`subroutine, public rich_ops::stop_rich_lvl`

Stop a Richardson level.

It decides whether convergence has been reached, and possibly sets a guess for the next level.

Definition at line 404 of file rich_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.31.2.7 term_rich()

```
subroutine, public rich_ops::term_rich
```

Terminate the Richardson extrapolation system.

Needs to be called only once.

Definition at line 219 of file rich_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.32 rich_vars Module Reference

Variables concerning Richardson extrapolation.

Functions/Subroutines

- elemental character(len=max_str_ln) function, public rich_info ()

  Returns string with possible extension with Richardson level or nothing if only one level and one parallel job.

Variables

- integer, public rich_lvl

  current level of Richardson extrapolation
- integer, public n_par_x

  nr. of parallel points in field-aligned grid
- integer, parameter, public req_min_n_par_x = 20

  required minimum n_par_X
- integer, public min_n_par_x

  min. of n_par_X (e.g. first value in Richardson loop)
- logical, public use_guess

  whether a guess is formed from previous level of Richardson
- logical, public no_guess

  disable guessing Eigenfunction from previous level of Richardson
- logical, public rich_conv

  if Richarson extrapolation has converged
- complex(dp), dimension(:,:,:), allocatable, public sol_val_rich

  Richardson array of eigenvalues.
- real(dp), dimension(:,:), allocatable, public x_axis_rich

  x axis for plot of Eigenvalues with Richardson level
- real(dp), dimension(:), allocatable, public max_rel_err

  maximum relative error for all Richardson levels
- integer, dimension(:,:), allocatable, public loc_max_rel_err

  location of maximum of relative error

B.32.1 Detailed Description

Variables concerning Richardson extrapolation.

B.32.2 Function/Subroutine Documentation

B.32.2.1 rich_info()

```
elemental character(len=max_str_ln) function, public rich_vars::rich_info
```

Returns string with possible extension with Richardson level or nothing if only one level and one parallel job.

Definition at line 34 of file rich_vars.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.32.3 Variable Documentation

B.32.3.1 loc_max_rel_err

```
integer, dimension(:,:), allocatable, public rich_vars::loc_max_rel_err
```

location of maximum of relative error

Definition at line 29 of file rich_vars.f90.

B.32.3.2 max_rel_err

```
real(dp), dimension(:), allocatable, public rich_vars::max_rel_err
```

maximum relative error for all Richardson levels

Definition at line 28 of file rich_vars.f90.

B.32.3.3 min_n_par_x

```
integer, public rich_vars::min_n_par_x
```

min. of `n_par_X` (e.g. first value in Richardson loop)

Definition at line 22 of file rich_vars.f90.

B.32.3.4 n_par_x

```
integer, public rich_vars::n_par_x
```

nr. of parallel points in field-aligned grid

Definition at line 20 of file rich_vars.f90.

B.32.3.5 no_guess

```
logical, public rich_vars::no_guess
```

disable guessing Eigenfunction from previous level of Richardson

Definition at line 24 of file rich_vars.f90.

B.32.3.6 req_min_n_par_x

```
integer, parameter, public rich_vars::req_min_n_par_x = 20
```

required minimum `n_par_X`

Definition at line 21 of file rich_vars.f90.

B.32.3.7 rich_conv

```
logical, public rich_vars::rich_conv
```

if Richarson extrapolation has converged

Definition at line 25 of file rich_vars.f90.

B.32.3.8 rich_lvl

```
integer, public rich_vars::rich_lvl
```

current level of Richardson extrapolation

Definition at line 19 of file rich_vars.f90.

B.32.3.9 sol_val_rich

```
complex(dp), dimension(:,:,:), allocatable, public rich_vars::sol_val_rich
```

Richardson array of eigenvalues.

Definition at line 26 of file rich_vars.f90.

B.32.3.10 use_guess

```
logical, public rich_vars::use_guess
```

whether a guess is formed from previous level of Richardson

Definition at line 23 of file rich_vars.f90.

B.32.3.11 x_axis_rich

```
real(dp), dimension(:,:), allocatable, public rich_vars::x_axis_rich
```

x axis for plot of Eigenvalues with Richardson level

Definition at line 27 of file rich_vars.f90.

B.33 slepc_ops Module Reference

Operations that use SLEPC (and PETSC) routines.

Functions/Subroutines

- integer function, public solve_ev_system_slepc (mds, grid_sol, x, vac, sol)

  Solve the eigenvalue system using SLEPC.
- integer function, public start_slepc (n_r_sol)

  This subroutine starts PETSC and SLEPC with the correct number of processors.
- integer function, public setup_mats (mds, grid_sol, x, a, b)

  Sets up the matrices $\overline{A}$ and $\overline{B}$ in the EV problem $\overline{A}\vec{X} = \lambda\overline{B}\vec{X}$..
- integer function, public set_bc (mds, grid_sol, x, vac, a, b)

  Sets the boundary conditions deep in the plasma (1) and at the edge (2).
- integer function, public setup_solver (x, a, b, solver)

  Sets up EV solver.
- integer function, public setup_guess (sol, a, solver)

  Sets up guess in solver.
- integer function, public get_solution (solver)

  Get the solution vectors.
- integer function, public summarize_solution (solver, max_n_ev)

  Summarizes solution.
- integer function, public store_results (mds, grid_sol, sol, solver, max_n_ev, a, b)

  Stores the results.
- integer function, public stop_slepc (a, b, solver)

  Stop PETSC and SLEPC.

Variables

- logical, public debug_setup_mats = .false.

  plot debug information for setup_mats
- logical, public debug_set_bc = .false.

  plot debug information for set_BC
- logical, public test_diff = .false.

  test introduction of numerical diff

B.33.1 Detailed Description

Operations that use SLEPC (and PETSC) routines.

Note

The routines here require a **TRIMMED** solution grid!

See also

References: [7]

B.33.2 Function/Subroutine Documentation

B.33.2.1 get_solution()

```
integer function, public slepc_ops::get_solution (
                intent(inout) solver)
```

Get the solution vectors.

Returns

ierr

Definition at line 1533 of file SLEPC_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.33.2.2 set_bc()

```
integer function, public slepc_ops::set_bc (
                type(modes_type), intent(in) mds,
                type(grid_type), intent(in) grid_sol,
                type(x_2_type), intent(in) x,
                type(vac_type), intent(in) vac,
                intent(inout) a,
                intent(inout) b)
```

Sets the boundary conditions deep in the plasma (1) and at the edge (2).

Through `norm_disc_style_sol` it can be chosen whether the finite differences are:

1. central finite differences
2. left finite differences

Possibilities for `BC_style`:

1. Set to zero:
   - An artificial Eigenvalue `EV_BC` is introduced by setting the diagonal components of A to EV_BC and of B to 1, and the off-diagonal elements to zero.
2. Minimization of surface energy through asymmetric fin. differences:
   - For symmetric finite differences, the last `ndps` grid points are treated asymmetrically in order not go over the edge and the vacuum term is added to the edge element.
   - For left differences, this is already standard, so the method becomes identical to 2.
3. Minimization of surface energy through extension of grid:
   - For symmetric finite differences, `ndps` extra grid points are introduced after the edge and the vacuum term is added to the edge element.
   - For left finite differences, the vacuum term is just added to the edge element, so this method becomes idential to 3.
4. Explicit introduction of the surface energy minimization:
   - The equation due to the minimization of the vacuum term is introduced explicitely as an asymmetric finite difference equation in the last row. -This is done using left finite differences.

Makes use of n_r.

Returns

ierr

Parameters

`in`	*mds*	general modes variables
`in`	*grid_sol*	solution grid
`in`	*x*	field-averaged perturbation variables (so only first index)
`in`	*vac*	vacuum variables

Definition at line 932 of file SLEPC_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.33.2.3 setup_guess()

```
integer function, public slepc_ops::setup_guess (
                type(sol_type), intent(in) sol,
                intent(in) a,
                intent(inout) solver)
```

Sets up guess in solver.

Returns

ierr

Parameters

in	*sol*	solution variables

Definition at line 1452 of file SLEPC_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.33.2.4 setup_mats()

```
integer function, public slepc_ops::setup_mats (
                type(modes_type), intent(in) mds,
                type(grid_type), intent(in) grid_sol,
                type(x_2_type), intent(in), target x,
                intent(inout) a,
                intent(inout) b)
```

Sets up the matrices $\overline{\mathrm{A}}$ and $\overline{\mathrm{B}}$ in the EV problem $\overline{\mathrm{A}}\vec{X} = \lambda\overline{\mathrm{B}}\vec{X}$..

The matrices are set up in the solution grid, so some processes might not have any part in the storage thereof (if `sol_n_procs` $<$ `n_procs`), but each process sets up the part of the grid that corresponds to their own normal range in the solution grid.

Returns

ierr

Parameters

in	*mds*	general modes variables
in	*grid_sol*	solution grid
in	*x*	field-averaged perturbation variables (so only first index)

Definition at line 408 of file SLEPC_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.33.2.5 setup_solver()

```
integer function, public slepc_ops::setup_solver (
                type(x_2_type), intent(in) x,
                intent(in) a,
                intent(in) b,
                intent(inout) solver)
```

Sets up EV solver.

Returns

ierr

Parameters

in	*x*	field-averaged perturbation variables (so only first index)

Definition at line 1325 of file SLEPC_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.33.2.6 solve_ev_system_slepc()

```
integer function, public slepc_ops::solve_ev_system_slepc (
                type(modes_type), intent(in) mds,
                type(grid_type), intent(in) grid_sol,
                type(x_2_type), intent(in) x,
                type(vac_type), intent(in) vac,
                type(sol_type), intent(inout) sol)
```

Solve the eigenvalue system using SLEPC.

This subroutine sets up the matrices A ad B of the generalized EV system described in [17] and solves them using the SLEPC suite. The solutions closest to a target indicated by EV_guess are obtained.

See also

read_input_opts()

Returns

ierr

Parameters

in	*mds*	general modes variables
in	*grid_sol*	solution grid
in	*x*	field-averaged perturbation variables (so only first index)
in	*vac*	vacuum variables
in,out	*sol*	solution variables

Definition at line 60 of file SLEPC_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.33.2.7 start_slepc()

```
integer function, public slepc_ops::start_slepc (
            integer, intent(in) n_r_sol)
```

This subroutine starts PETSC and SLEPC with the correct number of processors.

Returns

ierr

Parameters

in	*n_r_sol*	nr. of normal points in solution grid

Definition at line 332 of file SLEPC_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.33.2.8 stop_slepc()

```
integer function, public slepc_ops::stop_slepc (
                intent(in) a,
                intent(in) b,
                intent(in) solver)
```

Stop PETSC and SLEPC.

Returns

ierr

Definition at line 2053 of file SLEPC_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.33.2.9 store_results()

```
integer function, public slepc_ops::store_results (
                type(modes_type), intent(in) mds,
                type(grid_type), intent(in) grid_sol,
                type(sol_type), intent(inout) sol,
                intent(inout) solver,
                intent(inout) max_n_ev,
                intent(inout) a,
                intent(inout) b)
```

Stores the results.

Returns

ierr

Parameters

in	*mds*	general modes variables

in	*grid_sol*	solution grid
in,out	*sol*	solution variables

Definition at line 1632 of file SLEPC_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.33.2.10 summarize_solution()

```
integer function, public slepc_ops::summarize_solution (
                intent(inout) solver,
                intent(inout) max_n_ev)
```

Summarizes solution.

Returns

ierr

Definition at line 1563 of file SLEPC_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.33.3 Variable Documentation

B.33.3.1 debug_set_bc

```
logical, public slepc_ops::debug_set_bc = .false.
```

plot debug information for set_BC

Note

Debug version only

Definition at line 46 of file SLEPC_ops.f90.

B.33.3.2 debug_setup_mats

```
logical, public slepc_ops::debug_setup_mats = .false.
```

plot debug information for setup_mats

Note

Debug version only

Definition at line 45 of file SLEPC_ops.f90.

B.33.3.3 test_diff

```
logical, public slepc_ops::test_diff = .false.
```

test introduction of numerical diff

Note

Debug version only

Definition at line 47 of file SLEPC_ops.f90.

B.34 slepc_utilities Module Reference

Numerical utilities related to SLEPC (and PETSC) operations.

Functions/Subroutines

- integer function, public insert_block_mat (mds, block, mat, r_id, ind, n_r, transp, overwrite, ind_insert)
  Insert a block pertaining to 1 normal point into a matrix.

Variables

- logical, public debug_insert_block_mat = .false.
  plot debug information for insert_block_mat()

B.34.1 Detailed Description

Numerical utilities related to SLEPC (and PETSC) operations.

See also

References: [7]

B.34.2 Function/Subroutine Documentation

B.34.2.1 insert_block_mat()

```
integer function, public slepc_utilities::insert_block_mat (
                type(modes_type), intent(in), target mds,
                dimension(:,:), intent(in) block,
                intent(inout) mat,
                intent(in) r_id,
                dimension(2), intent(in) ind,
                intent(in) n_r,
                optional transp,
                optional overwrite,
                optional ind_insert)
```

Insert a block pertaining to 1 normal point into a matrix.

Optionally, also set the Hermitian transpose and / or overwrite instead of add value.

This routine takes into account that if the mode numbers change as a function of the normal coordinate, as is the case for X_style 2 (fast), part of the local blocks, which were set up without taking this into account, can have information that correspond to mode numbers that are not present any more for off-diagonal entries, and lack an equal amount of information that corresponds to the mode numbers that have taken their places.

Omission of this effect is justifiable if there are many more mode couplings that are not omitted (i.e. when `n_mod_X` is large), so that it is a small effect, and/or when the coupling between these modes is already negligible.

Unfortunately, it can be argued that this is not generally the case for the fast version (`X_style` = 2) since $\widetilde{PV}^1$ and $\widetilde{PV}^2$ are both $\sim (nq - m)$, which indicates that the smallest elements in the corresponding off-diagonal blocks lie in the central columns for $\widetilde{PV}^1$ and in the central columns and rows for $\widetilde{PV}^2$, while the problematic elements lie farthest away from the central columns for $\widetilde{PV}^1$ and the central columns and rows for $\widetilde{PV}^2$, so these are generally not small.

The coefficients due to normal discretization get increasingly small for higher orders, though, which compensates this for higher orders.

Finally, an alternative would be to set the problematic elements not to zero but extrapolated from the closest elements, but then consistency would be sacrified, for example in the energy reconstruction of the POST-processing: In the energy reconstruction the exact same terms are neglected as well.

To conclude, for fast `X_style` 2, the number of modes (`n_mod_X`) has to be chosen high enough compared with the variation of the mode numbers; i.e. the variation of the safety factor or rotational transform.

Note

> The grid in which the modes variables `mds` are tabulated up must be the same as the one in which `mat` is set up. If this is not the case, the indices in `mds` have to be adapted.

Returns

ierr

Parameters

in	*mds*	general modes variables

Definition at line 81 of file SLEPC_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.34.3 Variable Documentation

B.34.3.1 debug_insert_block_mat

```
logical, public slepc_utilities::debug_insert_block_mat = .false.
```

plot debug information for insert_block_mat()

Note

Debug version only

Definition at line 31 of file SLEPC_utilities.f90.

B.35 sol_ops Module Reference

Operations on the solution vectors such as decompososing the energy, etc...

Functions/Subroutines

- subroutine, public plot_sol_vals (sol, last_unstable_id)

  Plots Eigenvalues.
- integer function, public plot_sol_vec (mds_x, mds_sol, grid_eq, grid_x, grid_sol, eq_1, eq_2, x, sol, xyz, x_id, plot_var)

  Plots Eigenvectors.
- integer function, public plot_harmonics (mds, grid_sol, sol, x_id, res_surf)

  Plots the harmonics and their maximum in 2-D.
- integer function, public decompose_energy (mds_x, mds_sol, grid_eq, grid_x, grid_sol, eq_1, eq_2, x, sol, vac, x_id, b_aligned, xyz, e_pot_int, e_kin_int)

  Decomposes the plasma potential and kinetic energy in its individual terms for an individual Eigenvalue.
- integer function, public print_output_sol (grid, sol, data_name, rich_lvl, remove_previous_arrs)

  Print solution quantities to an output file:

Variables

- logical, public debug_plot_sol_vec = .false.

  plot debug information for plot_sol_vec()
- logical, public debug_calc_e = .false.

  plot debug information for calc_E()
- logical, public debug_x_norm = .false.

  plot debug information `X_norm`
- logical, public debug_du = .false.

  plot debug information for calculation of `DU`

B.35.1 Detailed Description

Operations on the solution vectors such as decompososing the energy, etc...

B.35.2 Function/Subroutine Documentation

B.35.2.1 decompose_energy()

```
integer function, public sol_ops::decompose_energy (
                type(modes_type), intent(in) mds_x,
                type(modes_type), intent(in) mds_sol,
                type(grid_type), intent(in) grid_eq,
                type(grid_type), intent(in) grid_x,
                type(grid_type), intent(in) grid_sol,
                type(eq_1_type), intent(in) eq_1,
                type(eq_2_type), intent(in) eq_2,
```

```
type(x_1_type), intent(in) x,
type(sol_type), intent(in) sol,
type(vac_type), intent(in) vac,
integer, intent(in) x_id,
logical, intent(in) b_aligned,
real(dp), dimension(:,:,:,:), intent(in), optional xyz,
complex(dp), dimension(7), intent(inout), optional e_pot_int,
complex(dp), dimension(2), intent(inout), optional e_kin_int)
```

Decomposes the plasma potential and kinetic energy in its individual terms for an individual Eigenvalue.

Use is made of variables representing the potential and kinetic energy [15].

- E_{pot}:
  - normal line bending term: $\frac{1}{\mu_0}\frac{Q_n^2}{h^{22}}$,
  - geodesic line bending term: $\frac{1}{\mu_0}\mathcal{J}^2\frac{h^{22}}{g_{33}}Q_g^2$,
  - normal ballooning term: $-2p'X^2\kappa_n$,
  - geodesic ballooning term: $-2p'XU^*\kappa_g$,
  - normal kink term: $-\sigma X^* Q_g$,
  - geodesic kink term: $\sigma U^* Q_n$,
- E_{kin}:
  - normal kinetic term: $\rho\frac{X^2}{h^{22}}$,
  - geodesic kinetic term: $\rho\mathcal{J}^2\frac{h^{22}}{g_{33}}U^2$.

The energy terms are calculated normally on the sol grid, interpolating the quantities defined on the eq grid, and angularly in the eq grid.

Optionally, the results can be plotted by providing X, Y and Z. By default, they are instead written to an output file.

Also, the fraction between potential and kinetic energy can be returned, compared with the eigenvalue.

Returns

ierr

Parameters

in	*mds_x*	general modes variables in pertubation grid
in	*mds_sol*	general modes variables in solution grid
in	*grid_eq*	equilibrium grid
in	*grid_x*	perturbation grid
in	*grid_sol*	solution grid

in	*eq_1*	flux equilibrium
in	*eq_2*	metric equilibrium
in	*x*	perturbation variables
in	*sol*	solution variables
in	*vac*	vacuum variables
in	*x_id*	nr. of Eigenvalue
in	*b_aligned*	whether grid is field-aligned
in	*xyz*	X, Y and Z for plotting
in,out	*e_pot_int*	integrated potential energy
in,out	*e_kin_int*	integrated kinetic energy

Definition at line 988 of file sol_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.35.2.2 plot_harmonics()

```
integer function, public sol_ops::plot_harmonics (
                type(modes_type), intent(in) mds,
                type(grid_type), intent(in) grid_sol,
                type(sol_type), intent(in) sol,
                integer, intent(in) x_id,
                real(dp), dimension(:,:), intent(in) res_surf)
```

Plots the harmonics and their maximum in 2-D.

Returns

ierr

Parameters

in	*mds*	general modes variables in solution grid
in	*grid_sol*	solution grid
in	*sol*	solution variables
in	*x_id*	nr. of Eigenvalue (for output name)
in	*res_surf*	resonant surfaces

Definition at line 645 of file sol_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.35.2.3 plot_sol_vals()

```
subroutine, public sol_ops::plot_sol_vals (
                type(sol_type), intent(in) sol,
                integer, intent(in) last_unstable_id)
```

Plots Eigenvalues.

Parameters

in	*sol*	solution variables
in	*last_unstable_id*	index of last unstable EV

Definition at line 36 of file sol_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.35.2.4 plot_sol_vec()

```
integer function, public sol_ops::plot_sol_vec (
                type(modes_type), intent(in) mds_x,
                type(modes_type), intent(in) mds_sol,
                type(grid_type), intent(in) grid_eq,
                type(grid_type), intent(in) grid_x,
                type(grid_type), intent(in) grid_sol,
                type(eq_1_type), intent(in) eq_1,
                type(eq_2_type), intent(in) eq_2,
                type(x_1_type), intent(in) x,
                type(sol_type), intent(in) sol,
```

```
real(dp), dimension(:,:,:,:), intent(in) xyz,
integer, intent(in) x_id,
logical, dimension(2), intent(in) plot_var)
```

Plots Eigenvectors.

This is done using the angular part of the the provided equilibrium grid and the normal part of the provided solution grid.

The perturbation grid is assumed to have the same angular coordinates as the equilibrium grid, and the normal coordinates correspond to either the equilibrium grid (`X_grid_style` 1), the solution grid (`X_grid_style` 2) or the enriched equilibrium grid (`X_grid_style` 3).

The output grid, furthermore, has the angular part corresponding to the equilibrium grid, and the normal part to the solution grid.

Note

The normalization factors are taken into account and the output is transformed back to unnormalized values:

$$\vec{\xi} \sim \frac{X_0}{R_0 B_0} \,,$$
$$\vec{Q} \sim \frac{X_0}{R_0^2} \,,$$

which translates to

$$X \sim X_0 \,,$$
$$U \sim \frac{X_0}{R_0^2 B_0} \,,$$
$$Qn \sim \frac{X_0 B_0}{R_0} \,,$$
$$Qg \sim \frac{X_0}{R_0^3} \,,$$

where X_0 is not determined but is common to all factors.

Returns

ierr

Parameters

in	*mds_x*	general modes variables in perturbation grid
in	*mds_sol*	general modes variables in solution grid
in	*grid_eq*	equilibrium grid
in	*grid_x*	perturbation grid
in	*grid_sol*	solution grid
in	*eq_1*	flux equilibrium

in	*eq_2*	metric equilibrium
in	*x*	perturbation variables
in	*sol*	solution variables
in	*xyz*	X, Y and Z of extended eq_grid
in	*x_id*	nr. of Eigenvalue (for output name)
in	*plot_var*	whether variables are plotted (1: plasma perturbation, 2: magnetic perturbation)

Definition at line 114 of file sol_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.35.2.5 print_output_sol()

```
integer function, public sol_ops::print_output_sol (
                type(grid_type), intent(in) grid,
                type(sol_type), intent(in) sol,
                character(len=*), intent(in) data_name,
                integer, intent(in), optional rich_lvl,
                logical, intent(in), optional remove_previous_arrs)
```

Print solution quantities to an output file:

- sol:
  - `val`
  - `vec`

If `rich_lvl` is provided, "`_[R_rich_lvl]`" is appended to the data name if it is $>$`0`.

Returns

ierr

Parameters

in	*grid*	solution grid variables
in	*sol*	solution variables
in	*data_name*	name under which to store
in	*rich_lvl*	Richardson level to print
in	*remove_previous_arrs*	remove previous variables if present

Definition at line 1564 of file sol_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.35.3 Variable Documentation

B.35.3.1 debug_calc_e

```
logical, public sol_ops::debug_calc_e = .false.
```

plot debug information for calc_E()

Note

Debug version only

Definition at line 29 of file sol_ops.f90.

B.35.3.2 debug_du

```
logical, public sol_ops::debug_du = .false.
```

plot debug information for calculation of `DU`

Note

Debug version only

Definition at line 31 of file sol_ops.f90.

B.35.3.3 debug_plot_sol_vec

```
logical, public sol_ops::debug_plot_sol_vec = .false.
```

plot debug information for plot_sol_vec()

Note

Debug version only

Definition at line 27 of file sol_ops.f90.

B.35.3.4 debug_x_norm

```
logical, public sol_ops::debug_x_norm = .false.
```

plot debug information `X_norm`

Note

Debug version only

Definition at line 30 of file sol_ops.f90.

B.36 sol_utilities Module Reference

Numerical utilities related to the solution vectors.

Interfaces and Types

- interface calc_xuq

  Calculates the normal ($\cdot_n$) or geodesic ($\cdot_g$) components of the plasma perturbation $\vec{\xi}$ or the magnetic perturbation $\vec{Q} = \nabla \times \left(\vec{x} \times \vec{B}\right)$.

Functions/Subroutines

- integer function, public calc_tot_sol_vec (mds, grid_sol, sol_vec_loc, sol_vec_tot, deriv)

  Calculate the total version of the solution vector from the local version.
- integer function, public calc_loc_sol_vec (mds, i_min, sol_vec_tot, sol_vec_loc)

  Calculate the local version of the solution vector from the total version.
- integer function, public interp_v_spline (v_i, v_o, r_i, r_o, extrap, ivs_stat)

  Interpolation for a quantity V using splines.

Variables

- logical, public debug_calc_xuq_arr = .false.

  plot debug information for calc_XUQ_arr
- logical, public debug_interp_v_spline = .false.

  debug information for interp_v_spline

B.36.1 Detailed Description

Numerical utilities related to the solution vectors.

B.36.2 Function/Subroutine Documentation

B.36.2.1 calc_loc_sol_vec()

```
integer function, public sol_utilities::calc_loc_sol_vec (
                type(modes_type), intent(in) mds,
                integer, intent(in) i_min,
                complex(dp), dimension(:,:), intent(in) sol_vec_tot,
                complex(dp), dimension(:,:), intent(inout), allocatable sol_vec_loc)
```

Calculate the local version of the solution vector from the total version.

See also

See calc_tot_sol_vec().

Returns

ierr

Parameters

in	*mds*	general modes variables

in	*i_min*	i_min of grid in which variables are tabulated
in	*sol_vec_tot*	solution vector for all possible resonating modes
in,out	*sol_vec_loc*	solution vector for local resonating modes

Definition at line 720 of file sol_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.36.2.2 calc_tot_sol_vec()

```
integer function, public sol_utilities::calc_tot_sol_vec (
                type(modes_type), intent(in) mds,
                type(grid_type), intent(in) grid_sol,
                complex(dp), dimension(:,:), intent(in) sol_vec_loc,
                complex(dp), dimension(:,:), intent(inout), allocatable sol_vec_tot,
                integer, intent(in), optional deriv)
```

Calculate the total version of the solution vector from the local version.

This is the solution vector for all of the possible mode numbers, which can be different from the local mode numbers for X style 2 (fast):

- local: the size of the saved perturbation and solution variables, prescribed by the user as input:
  - either directly through n_mod_X

  - or through the limits on the modes using `min_sec_X` and `max_sec_X`.
- total: all the possible modes that can resonate in the plasma, which can be different from the local number for `X_style` 2, as the mode numbers depend on the normal coordinate.

If the output variable is not allocated, it is done here.

Optionally a derivative can be requested, depending on the normal discretization order.

Note

Need to provide the grid in which mds is tabulated.

Returns

ierr

Parameters

`in`	*mds*	general modes variables
`in`	*grid_sol*	solution grid
`in`	*sol_vec_loc*	solution vector for local resonating modes
`in,out`	*sol_vec_tot*	solution vector for all possible resonating modes
`in`	*deriv*	order of derivative

Definition at line 579 of file sol_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.36.2.3 interp_v_spline()

```
integer function, public sol_utilities::interp_v_spline (
                complex(dp), dimension(:,:,:), intent(in) v_i,
                complex(dp), dimension(:,:,:), intent(out) v_o,
                real(dp), dimension(:), intent(in) r_i,
                real(dp), dimension(:), intent(in) r_o,
                logical, intent(in) extrap,
                integer, intent(out), optional ivs_stat)
```

Interpolation for a quantity V using splines.

Optionally, a variable 'ivs_stat' is returned that indicates whether the interpolation was a plain copy (1), a linear interpolation (2) or a spline interpolation (3).

Definition at line 799 of file sol_utilities.f90.

Here is the caller graph for this function:

B.36.3 Variable Documentation

B.36.3.1 debug_calc_xuq_arr

```
logical, public sol_utilities::debug_calc_xuq_arr = .false.
```

plot debug information for calc_XUQ_arr

Note

Debug version only

Definition at line 25 of file sol_utilities.f90.

B.36.3.2 debug_interp_v_spline

```
logical, public sol_utilities::debug_interp_v_spline = .false.
```

debug information for interp_v_spline

Note

Debug version only

Definition at line 26 of file sol_utilities.f90.

B.37 sol_vars Module Reference

Variables pertaining to the solution quantities.

Interfaces and Types

- type sol_type
  solution type

Functions/Subroutines

- subroutine init_sol (sol, mds, grid_sol, n_ev, lim_sec_x)
  Initialize a solution type and allocate the variables.
- subroutine dealloc_sol (sol)
  Deallocates solution variables.

Variables

- integer, public n_alloc_sols
  nr. of allocated grids

B.37.1 Detailed Description

Variables pertaining to the solution quantities.

B.37.2 Function/Subroutine Documentation

B.37.2.1 dealloc_sol()

```
subroutine sol_vars::dealloc_sol (
              class(sol_type), intent(inout) sol)
```

Deallocates solution variables.

Note

intent(out) automatically deallocates the variable

Parameters

in,out	*sol*	solution variables to be deallocated

Definition at line 105 of file sol_vars.f90.

Here is the call graph for this function:

B.37.2.2 init_sol()

```
subroutine sol_vars::init_sol (
            class(sol_type), intent(inout) sol,
            type(modes_type), intent(in) mds,
            type(grid_type), intent(in) grid_sol,
            integer, intent(in) n_ev,
            integer, dimension(2), intent(in), optional lim_sec_x)
```

Initialize a solution type and allocate the variables.

The number of modes as well as `n` and `m` are also set up.

Optionally, the secondary mode number can be specified (`m` if poloidal flux is used as normal coordinate and c n if toroidal flux). By default, they are taken from the global `X_vars` variables.

Note

If the lowest limits of the grid is not 1 (e.g. `grid_soli_min = 1` for first process), the input variable `i_min` should be set to set correctly. For a full grid, it should be set to 1.

Parameters

in,out	*sol*	solution variables
in	*mds*	general modes variables
in	*grid_sol*	solution grid

in	*n_ev*	nr. of Eigenvalues
in	*lim_sec↩ _x*	limits of m_X (pol. flux) or n_X (tor. flux)

Definition at line 59 of file sol_vars.f90.

Here is the call graph for this function:

B.37.3 Variable Documentation

B.37.3.1 n_alloc_sols

```
integer, public sol_vars::n_alloc_sols
```

nr. of allocated grids

Note

Debug version only

Definition at line 22 of file sol_vars.f90.

B.38 str_utilities Module Reference

Operations on strings.

Functions/Subroutines

- elemental character(len=max_str_ln) function, public i2str (k)
  Convert an integer to string.
- elemental character(len=max_str_ln) function, public ii2str (k)
  Convert an integer to string.
- elemental character(len=max_str_ln) function, public r2str (k)
  Convert a real (double) to string.
- elemental character(len=max_str_ln) function, public r2strt (k)
  Convert a real (double) to string.
- elemental character(len=max_str_ln) function, public c2str (k)
  Convert a complex (double) to string.
- elemental character(len=max_str_ln) function, public c2strt (k)
  Convert a complex (double) to string.
- character(len(input_string)) function, public strh2l (input_string)
  Convert a string to lowercase.
- character(len(input_string)) function, public strl2h (input_string)
  convert a string to uppercase.
- character((len(input_strings)+2) ∗size(input_strings)) function, public merge_strings (input_strings)
  Merge array of strings.

B.38.1 Detailed Description

Operations on strings.

B.38.2 Function/Subroutine Documentation

B.38.2.1 c2str()

```
elemental character(len=max_str_ln) function, public str_utilities::c2str (
            complex(dp), intent(in) k)
```

Convert a complex (double) to string.

Note

See http://www.fortran90.org/src/best-practices.html how to not lose precision.

Parameters

in	*k*	complex to convert

Definition at line 65 of file str_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.38.2.2 c2strt()

```
elemental character(len=max_str_ln) function, public str_utilities::c2strt (
            complex(dp), intent(in) k)
```

Convert a complex (double) to string.

Version with less precise output.

See also

See c2str().

Parameters

in	k	complex to convert

Definition at line 87 of file str_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.38.2.3 i2str()

```
elemental character(len=max_str_ln) function, public str_utilities::i2str (
            integer, intent(in) k)
```

Convert an integer to string.

See also

from http://stackoverflow.com/questions/1262695/converting-integers-to-strings-in-fortran

Parameters

in	*k*	integer to convert

Definition at line 17 of file str_utilities.f90.

Here is the call graph for this function:

B.38.2.4 ii2str()

```
elemental character(len=max_str_ln) function, public str_utilities::ii2str (
            integer(kind=8), intent(in) k)
```

Convert an integer to string.

Version with kind 8 integers.

See also

See i2str().

Parameters

in	*k*	integer to convert

Definition at line 29 of file str_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.38.2.5 merge_strings()

```
character((len(input_strings)+2)*size(input_strings)) function, public str_utilities::merge_strings (
            character(*), dimension(:), intent(in) input_strings)
```

Merge array of strings.

Parameters

in	*input_strings*	array of strings

Returns

merged string

Definition at line 151 of file str_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.38.2.6 r2str()

```
elemental character(len=max_str_ln) function, public str_utilities::r2str (
            real(dp), intent(in) k)
```

Convert a real (double) to string.

Note

See http://www.fortran90.org/src/best-practices.html how to not lose precision.

Parameters

in	*k*	real to convert

Definition at line 41 of file str_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.38.2.7 r2strt()

```
elemental character(len=max_str_ln) function, public str_utilities::r2strt (
            real(dp), intent(in) k)
```

Convert a real (double) to string.

Version with less precise output.

See also

See r2str().

Parameters

in	*k*	real to convert

Definition at line 53 of file str_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.38.2.8 strh2l()

```
character(len(input_string)) function, public str_utilities::strh2l (
            character(*), intent(in) input_string)
```

Convert a string to lowercase.

See also

from [12], figure 3.5B, pg 80.

Parameters

in	*input_string*	input string

Returns

lowercase version

Definition at line 108 of file str_utilities.f90.

Here is the caller graph for this function:

B.38.2.9 strl2h()

```
character(len(input_string)) function, public str_utilities::strl2h (
            character(*), intent(in) input_string)
```

convert a string to uppercase.

See also

See strh2l()

Parameters

in	*input_string*	input string

Returns

uppercase version

Definition at line 130 of file str_utilities.f90.

B.39 test Module Reference

Generic tests.

Functions/Subroutines

- integer function, public generic_tests ()
  Performs generic tests.

B.39.1 Detailed Description

Generic tests.

B.39.2 Function/Subroutine Documentation

B.39.2.1 generic_tests()

```
integer function, public test::generic_tests
```

Performs generic tests.

These tests are run in interactive fashion by running the program with the `--test` command-line option.

See also

init_files()

Note

1. It is probably also recommended to run it with `--do_execute_command_line` in order to generate the plots in real time.
2. It has not been tested whether testing routines "pollutes" the further simulations. It is therefore best to not use the tests when doing a real run.

Debug version only

Returns

ierr

Definition at line 38 of file test.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.40 vac_ops Module Reference

Operations and variables pertaining to the vacuum response.

Functions/Subroutines

- integer function, public store_vac (grid, eq_1, eq_2, vac)
  Stores calculation of the vacuum response by storing the necessary variables.
- integer function calc_gh_1 (vac, n_r_in, lims_r_in, x_vec_in, g_in, h_in, ext_in)
  Calculates matrices `G` and `H` in 3-D configuration.
- integer function calc_gh_2 (vac, n_r_in, lims_r_in, x_vec_in, g_in, h_in, ext_in)
  Calculates matrices `G` and `H` in axisymmetric configurations.
- integer function, public calc_vac_res (mds, vac)
  Calculates the vacuum response.
- integer function, public vac_pot_plot (grid_sol, vac, sol, x_id)
  Calculate vacuum potential at some positions that are not resonant.
- integer function solve_phi_bem (vac, r, phi, n_rphi, n_loc_rphi, lims_c_rphi, desc_rphi)
  Calculates potential `Phi` on boundary of BEM in terms of some right-hand side .
- integer function, public print_output_vac (vac, data_name, rich_lvl)
  Print vacuum quantities of a certain order to an output file.

Variables

- logical, public debug_calc_gh = .false.
  plot debug information for calc_GH()
- logical, public debug_calc_vac_res = .false.
  plot debug information for calc_vac_res()
- logical, public debug_vac_pot_plot = .false.
  plot debug information for vac_pot_plot()

B.40.1 Detailed Description

Operations and variables pertaining to the vacuum response.

The vacuum response is calculated using the Boundary Element Method. There are two possibilities, indicated by the variable `style` in `vac_vars`.

1. 3-D field aligned:
   - Makes use of the collocation method for the grid points along the magnetic field lines.
   - The 3-D integrals are therefore integrated using Weyl's theorem [6] , p. 5.
2. axisymmetric:
   - Makes use of an analytical expression for the toroidal integration of the Green's functions, as shown in [3] .
   - The integration in the poloidal angle is done using the collocation method.

The final matrix equation is solved through Strumpack [10] .

See also

See [15] .

Todo The vacuum part of PB3D is still under construction and not usable yet.

B.40.2 Function/Subroutine Documentation

B.40.2.1 calc_gh_1()

```
integer function vac_ops::calc_gh_1 (
            type(vac_type), intent(inout) vac,
            integer, intent(in) n_r_in,
            integer, dimension(:,:), intent(in) lims_r_in,
            real(dp), dimension(:,:), intent(in) x_vec_in,
            real(dp), dimension(:,:), intent(in), pointer g_in,
            real(dp), dimension(:,:), intent(in), pointer h_in,
            logical, intent(in) ext_in)
```

Calculates matrices G and H in 3-D configuration.

See also

calc_gh.

Parameters

in,out	*vac*	vacuum variables
in	*n_r_in*	total number of rows of external poins of influence
in	*lims_r_in*	row limits of external points of influence
in	*x_vec_in*	position of influence
in	*g_in*	G at position of influence
in	*h_in*	H at position of influence
in	*ext_in*	position of influence is external

Definition at line 777 of file vac_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.40.2.2 calc_gh_2()

```
integer function vac_ops::calc_gh_2 (
                type(vac_type), intent(inout) vac,
                integer, intent(in) n_r_in,
                integer, dimension(:,:), intent(in) lims_r_in,
                real(dp), dimension(:,:), intent(in) x_vec_in,
                real(dp), dimension(:,:), intent(in), pointer g_in,
                real(dp), dimension(:,:), intent(in), pointer h_in,
                logical, intent(in) ext_in)
```

Calculates matrices G and H in axisymmetric configurations.

It makes use of toroidal functions.

See also

calc_gh.

Parameters

in,out	*vac*	vacuum variables
in	*n_r_in*	total number of rows of external poins of influence
in	*lims_r_in*	row limits of external points of influence
in	*x_vec_in*	position of influence
in	*g_in*	G at position of influence
in	*h_in*	H at position of influence
in	*ext_in*	position of influence is external

Definition at line 981 of file vac_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.40.2.3 calc_vac_res()

```
integer function, public vac_ops::calc_vac_res (
                type(modes_type), intent(in) mds,
                type(vac_type), intent(inout) vac)
```

Calculates the vacuum response.

First `G` and `H` are completed if this is not the first Richardson level.

Then the vacuum contribution is calculated using the two-fold strategy of first solving $\overline{\mathsf{H}}\overline{\Phi} = \overline{\mathsf{GEP}}$, followed by left-multiplication of $\overline{\Phi}$ by $\overline{\mathsf{PE}}^{\dagger}$.

The non-square matrix $\overline{\mathsf{E}}$ contains the exponents for different mode numbers and different poloidal grid points, whereas $\overline{\mathsf{P}}$ are diagonal matrices that contain the factors $(nq - m)$.

In practice, the complex matrix $\overline{\mathsf{E}}$ is split in the two components of a real matrix twice the width.

For vacuum style 1, the poloidal grid points correspond to the paralell grid points and have to be provied by a `grid` variable.

If `jump_to_sol` is used for the current Richardson level, the vacuum quantities are not calculated, but just restored.

Currently, this procedure only works for vacuum style 2 (axisymmetric).

Returns

ierr

Parameters

in	*mds*	general modes variables

in,out	*vac*	vacuum variables

Definition at line 1416 of file vac_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.40.2.4 print_output_vac()

```
integer function, public vac_ops::print_output_vac (
                type(vac_type), intent(in) vac,
                character(len=*), intent(in) data_name,
                integer, intent(in), optional rich_lvl)
```

Print vacuum quantities of a certain order to an output file.

- vac:
  - misc_vac
  - norm
  - x_vec
  - vac_res

If rich_lvl is provided, "_R_[rich_lvl]" is appended to the data name if it is >0.

Note

1. This routine should be called by a single process, in contrast to the other output routines such as eq_ops.print_output_eq(), print_output_sol(), ...
2. This process should be the last one, as it will set the boundary contribution.
3. This routine does not save the `H` and `G` matrices because they are large and foreseen to be saved directly as Hierarchical matrices in the future. They therefore need to be regenerated if Richardson restart is done, or when after a jump to solution, another Richardson level is attempted. In calc_vac, it is checked when copying the vacuum variables from previous to current Richardson level, whether the `G` and `H` matrices are allocated and if not, they are calculated.

Returns

ierr

Parameters

in	*vac*	vacuum variables
in	*data_name*	name under which to store
in	*rich_lvl*	Richardson level to print

Definition at line 2346 of file vac_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.40.2.5 solve_phi_bem()

```
integer function vac_ops::solve_phi_bem (
                type(vac_type), intent(in), target vac,
                real(dp), dimension(:,:), intent(in) r,
                real(dp), dimension(:,:), intent(inout), target phi,
                integer, dimension(2), intent(in) n_rphi,
                integer, dimension(2), intent(in) n_loc_rphi,
                integer, dimension(:,:), intent(in) lims_c_rphi,
                integer, dimension(blacsctxtsize), intent(in), target desc_rphi)
```

Calculates potential `Phi` on boundary of BEM in terms of some right-hand side .

This is done by solving with STRUMPack the system of equations $\overline{\mathrm{H}}\overline{\Phi} = \overline{\mathrm{G}}\overline{\mathrm{R}}$, where $\overline{\mathrm{R}}$ is the right-hand side, for example equal to $\overline{\mathrm{EP}}$ to solve in function of the individual Fourier modes.

Parameters

in	*vac*	vacuum variables
in,out	*phi*	$\overline{\Phi}$
in	*r*	$\overline{\mathrm{R}}$
in	*n_rphi*	n for $\overline{\mathrm{R}}$ and $\overline{\Phi}$
in	*n_loc_rphi*	local n for $\overline{\mathrm{R}}$ and $\overline{\Phi}$
in	*lims_c_rphi*	local limits for row and columns of $\overline{\mathrm{R}}$ and $\overline{\Phi}$
in	*desc_rphi*	descriptor for $\overline{\mathrm{R}}$ and $\overline{\Phi}$

Definition at line 2161 of file vac_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.40.2.6 store_vac()

```
integer function, public vac_ops::store_vac (
                type(grid_type), intent(in) grid,
                type(eq_1_type), intent(in) eq_1,
                type(eq_2_type), intent(in) eq_2,
                type(vac_type), intent(inout) vac)
```

Stores calculation of the vacuum response by storing the necessary variables.

This is done by the last process, which is the one that contains the variables at the plasma edge, and then this is broadcasted to the other processes.

The workings of this routine are summarized in the following diagram for VMEC:

Before storing the new variables, the procedure checks the following things:

- For the first equilibrium job, this routine copies the results from the previous Richardson level into the appropriate subranges of the new vacuum variables if no restart of Richardson levels was done.
- If a restart was done and the level is greater than one, there is a reconstruction of the previous level's variables. But this will happen in calc_vac_res(), not in this procedure.

If there is a jump straight to the solution, however, the procedure returns after reconstructing the current level's variables.

If the previous G and H variables are empty, they are regenerated later, in calc_gh(). This indicates that a reconstruction happened, either because a Richardson restart was performed, or because a new Richardson level was started after a previous level in which it was jumped straight to the solution.

For HELENA, there is only 1 equilibrium job at the first Richardson level, and the vacuum has to be calculated only once then. If there is a jump to solution or if there is Richardson restart, it only needs to be reconstructed.

Returns

ierr

Parameters

in	*grid*	equilibrium grid
in	*eq↩ _1*	flux equilibrium variables
in	*eq↩ _2*	metric equilibrium variables
in,out	*vac*	vacuum variables

Definition at line 112 of file vac_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.40.2.7 vac_pot_plot()

```
integer function, public vac_ops::vac_pot_plot (
                type(grid_type), intent(in) grid_sol,
                type(vac_type), intent(inout) vac,
                type(sol_type), intent(in) sol,
                integer, intent(in) x_id)
```

Calculate vacuum potential at some positions that are not resonant.

This is done by using the relation between H and G on the plasma boundary, $\overline{\mathrm{H}}_s\overline{\Phi}_s = \overline{\mathrm{G}}_s\overline{\mathrm{D}\phi}_s$, where $\overline{\mathrm{D}\phi}_s$ is the normal derivative of the potential on the plasma edge. The matrices $\overline{\mathrm{H}}$ and $\overline{\mathrm{G}}$ have to be calculated in advance.

This relation is then used at different positions to calculate the potential at a different position through $4\pi\phi = \left[-\overline{\mathrm{HIH}}_s^{-1}\overline{\mathrm{G}}_s + \overline{\mathrm{GI}}\right]\overline{\mathrm{D}\phi}_s$, where $\overline{\mathrm{I}}$ is an integration rule.

This system of equations contains one row per point at which the potential is to be calculated.

Currently, this routine creates a 3-D regular grid that is equidistant in cylindrical coordinates, with `n_vac_plot` values in R and Z, and `n_zeta_plot` values in the cylindrical angle. The limits in these variables, furthermore, are given by `min_Rvac_plot`, `max_Rvac_plot`, `min_Zvac_plot`, `max_Zvac_plot`, `min_zeta_plot` and `max_zeta_plot`.

Parameters

in	*grid_sol*	solution grid
in,out	*vac*	vacuum variables
in	*sol*	solution variables
in	*x_id*	nr. of Eigenvalue

Definition at line 1856 of file vac_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.40.3 Variable Documentation

B.40.3.1 debug_calc_gh

```
logical, public vac_ops::debug_calc_gh = .false.
```

plot debug information for calc_GH()

Note

Debug version only

Definition at line 46 of file vac_ops.f90.

B.40.3.2 debug_calc_vac_res

```
logical, public vac_ops::debug_calc_vac_res = .false.
```

plot debug information for calc_vac_res()

Note

Debug version only

Definition at line 47 of file vac_ops.f90.

B.40.3.3 debug_vac_pot_plot

```
logical, public vac_ops::debug_vac_pot_plot = .false.
```

plot debug information for vac_pot_plot()

Note

Debug version only

Definition at line 48 of file vac_ops.f90.

B.41 vac_utilities Module Reference

Numerical utilities related to the vacuum response.

Functions/Subroutines

- subroutine, public calc_gh_int_1 (g, h, x_s, x_in, norm_s, h_fac_in, step_size, tol)
  Calculate G_ij and H_ij on an interval for field line aligned configurations.
- subroutine, public calc_gh_int_2 (g, h, t_s, t_in, x_s, x_in, norm_s, norm_in, aij, ql, tol, b_coeff, n_tor)
  Calculate G_ij and H_ij on an interval for axisymmetric configurations.
- integer function, public vec_dis2loc (ctxt, vec_dis, lims, vec_loc, proc)
  Gathers a distributed vector on a single process.
- integer function, public mat_dis2loc (ctxt, mat_dis, lims_r, lims_c, mat_loc, proc)
  Gathers a distributed vector on a single process.
- integer function, public interlaced_vac_copy (vac_old, vac)
  Copies vacuum variables of a previous level into the appropriate, interlaced place of a next level.

B.41.1 Detailed Description

Numerical utilities related to the vacuum response.

See also

vac_ops for information.

B.41.2 Function/Subroutine Documentation

B.41.2.1 calc_gh_int_1()

```
subroutine, public vac_utilities::calc_gh_int_1 (
                real(dp), dimension(2), intent(inout) g,
                real(dp), dimension(2), intent(inout) h,
                real(dp), dimension(2,3), intent(in) x_s,
                real(dp), dimension(3), intent(in) x_in,
                real(dp), dimension(2,3), intent(in) norm_s,
                real(dp), dimension(4), intent(in) h_fac_in,
                real(dp), dimension(2), intent(in) step_size,
                real(dp), intent(in) tol)
```

Calculate G_ij and H_ij on an interval for field line aligned configurations.

The indices for the source variables are `[left:right,dim]` where `dim` is the Cartesian dimension. The same holds for `ql`.

For subintegrals close to the singularity, the procedure uses the analytical approximation.

Note

This routine does not calculate the contribution 2β.

Parameters

in	*x_s*	position vector at left and right limit of source interval

in	*x_in*	position vector at which to calculate influence
in	*norm_s*	normal vector at left and right limit of source interval
in	*h_fac_in*	metric factors at which to calculate influence
in	*step_size*	step sizes in parallel direction and alpha
in	*tol*	tolerance on distance between points

Definition at line 32 of file vac_utilities.f90.

Here is the caller graph for this function:

B.41.2.2 calc_gh_int_2()

```
subroutine, public vac_utilities::calc_gh_int_2 (
                real(dp), dimension(2), intent(inout) g,
                real(dp), dimension(2), intent(inout) h,
                real(dp), dimension(2), intent(in) t_s,
                real(dp), intent(in) t_in,
                real(dp), dimension(2,2), intent(in) x_s,
                real(dp), dimension(2), intent(in) x_in,
                real(dp), dimension(2,2), intent(in) norm_s,
                real(dp), dimension(2), intent(in) norm_in,
                real(dp), dimension(2), intent(in) aij,
                real(dp), dimension(2,2), intent(in) ql,
                real(dp), intent(in) tol,
                real(dp), dimension(2), intent(in) b_coeff,
                integer, intent(in) n_tor)
```

Calculate G_ij and H_ij on an interval for axisymmetric configurations.

The indices for the source variables are `[left:right,dim]` where `dim` is the Cartesian dimension. The same holds for `ql`.

For subintegrals close to the singularity (i.e. where the tolerance is no being met), the routine approximates the toroidal functions with the analytical approximation. If one of the boundaries of the subintegral is far enough from the singularity, there is an additional contribution due to the difference between the actual toroidal function and the approximation. This contribution is zero if both boundaries are close, because in this case it is assumed that the entire integral is given by the analytical approximation.

Note

This routine does not calculate the contribution 2β.

Parameters

in	*t_s*	pol. angle at left and right limit of source interval

in	*t_in*	pol. angle at which to calculate influence
in	*x_s*	position vector at left and right limit of source interval
in	*x_in*	position vector at which to calculate influence
in	*norm_s*	normal vector at left and right limit of source interval
in	*norm_↩ in*	normal vector at which to calculate influence
in	*aij*	Aij helper variable for H
in	*ql*	ql for n-1 and n at left and right limit of source interval
in	*tol*	tolerance on rho2
in	*b_coeff*	$b = \sum_{k=1}^{n} \frac{2}{2k-1}$ for n and n-1
in	*n_tor*	toroidal mode number

Definition at line 97 of file vac_utilities.f90.

Here is the caller graph for this function:

B.41.2.3 interlaced_vac_copy()

```
integer function, public vac_utilities::interlaced_vac_copy (
                type(vac_type), intent(in) vac_old,
                type(vac_type), intent(inout) vac)
```

Copies vacuum variables of a previous level into the appropriate, interlaced place of a next level.

This is done by multiplying left by $\overline{\mathsf{T}}$ and right by $\overline{\mathsf{T}}^T$ with

$$a_{ij} = \begin{cases} 1 & \text{for } (i,j) = (2j-1+(i-1)n_{\text{old}}, j+(i-1)n_{\text{old}}) \\ 0 & \text{otherwise} \end{cases}$$

where i ranges from 1 to n_{old} and j from 1 to m_{old}, with the size of $\overline{\mathsf{T}}$ equal to $n_{\text{new}} \times n_{\text{old}}$ where n refers to `n_bnd(1)` and m to `n_bnd(2)`.

Parameters

in	*vac_old*	old vacuum
in,out	*vac*	vacuum

Definition at line 354 of file vac_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.41.2.4 mat_dis2loc()

```
integer function, public vac_utilities::mat_dis2loc (
            integer, intent(in) ctxt,
            real(dp), dimension(:,:), intent(in) mat_dis,
            integer, dimension(:,:), intent(in) lims_r,
            integer, dimension(:,:), intent(in) lims_c,
            real(dp), dimension(:,:), intent(inout) mat_loc,
            integer, intent(in), optional proc)
```

Gathers a distributed vector on a single process.

See also

See vec_dis2loc() for exaplanation.

Parameters

in	*ctxt*	context for matrix
in	*mat_dis*	distributed matrix
in	*lims_r*	limits for different subrows
in	*lims_c*	limits for different subcolumns
in,out	*mat_loc*	local matrix

in	*proc*	which process receives result

Definition at line 275 of file vac_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.41.2.5 vec_dis2loc()

```
integer function, public vac_utilities::vec_dis2loc (
                integer, intent(in) ctxt,
                real(dp), dimension(:), intent(in) vec_dis,
                integer, dimension(:,:), intent(in) lims,
                real(dp), dimension(:), intent(inout) vec_loc,
                integer, intent(in), optional proc)
```

Gathers a distributed vector on a single process.

For the distributed vector, the limits of the different subrows need to be provided, as discussed in init_vac().

By default, the result lands on the master process, but this can be changed using `proc`. If it is negative, all processes receive the result. In any case, `vec_loc` will be utilized.

Note

vec_loc needs to be allocated to the total dimensions, even when the results is not received by the process.

Parameters

in	*ctxt*	context for vector

in	*vec_dis*	distributed vector
in	*lims*	limits for different subrows
in,out	*vec_loc*	local vector
in	*proc*	which process receives result

Definition at line 222 of file vac_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.42 vac_vars Module Reference

Variables pertaining to the vacuum quantities.

Interfaces and Types

- type vac_type
  vacuum type

Functions/Subroutines

- integer function init_vac (vac, style, n_bnd, prim_x, n_ang, jq)
  Initializes a vacuum type.
- subroutine, public set_loc_lims (n_loc, bs, ind_p, n_p, lims)
  Calculates the limits in local index.
- subroutine dealloc_vac (vac)
  Deallocates vacuum variables.
- integer function, public copy_vac (vac, vac_copy)
  Copy a vacuum type.
- logical function, public in_context (ctxt)
  Indicates whether current process is in the context.

Variables

- integer, public n_alloc_vacs
  nr. of allocated vacs

B.42.1 Detailed Description

Variables pertaining to the vacuum quantities.

B.42.2 Function/Subroutine Documentation

B.42.2.1 copy_vac()

```
integer function, public vac_vars::copy_vac (
                class(vac_type), intent(in) vac,
                class(vac_type), intent(inout) vac_copy)
```

Copy a vacuum type.

Note

The copy should be unallocated.

Parameters

in	*vac*	vac to be copied
in,out	*vac_copy*	copy

Definition at line 338 of file vac_vars.f90.

Here is the caller graph for this function:

B.42.2.2 dealloc_vac()

```
subroutine vac_vars::dealloc_vac (
            class(vac_type), intent(inout) vac)
```

Deallocates vacuum variables.

Parameters

in,out	*vac*	vacuum variables to be deallocated

Definition at line 286 of file vac_vars.f90.

Here is the call graph for this function:

B.42.2.3 in_context()

```
logical function, public vac_vars::in_context (
            integer, intent(in) ctxt)
```

Indicates whether current process is in the context.

Parameters

in	*ctxt*	context for vector

Definition at line 377 of file vac_vars.f90.

Here is the caller graph for this function:

B.42.2.4 init_vac()

```
integer function vac_vars::init_vac (
                class(vac_type), intent(inout) vac,
                integer, intent(in) style,
                integer, intent(in) n_bnd,
                integer, intent(in) prim_x,
                integer, dimension(2), intent(in) n_ang,
                real(dp), intent(in) jq)
```

Initializes a vacuum type.

The number of modes as well as `n` and `m` are also set up.

The variables G and H are saved in a special format, based on the block-cyclical distribution employed in scalapack. As a hypothetical example, consider a process grid corresponding to block-size 1x2 and 2x3 processes with a total size of 3x15:

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 \\ 3 & 3 & 4 & 4 & 5 & 5 & 3 & 3 & 4 & 4 & 5 & 5 & 3 & 3 & 4 \\ 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 \end{bmatrix}$$

Therefore, the 3-D storage convention used is, for example, for process 1:

$$\begin{bmatrix} (1,3) & (1,4) & (1,9) & (1,10) & (1,15) & (1,\cdot) \\ (3,3) & (3,4) & (3,9) & (3,10) & (3,15) & (3,\cdot) \end{bmatrix}$$

The information concerning the delimitations of the individual subintervals in the horizontal and vertical direction can be saved once per subinterval. For process 4 of above example, this would be

$$\begin{bmatrix} 2 \\ 2 \end{bmatrix};$$

for the vertical direction, and

$$\begin{bmatrix} 3 & 9 & 15 \\ 4 & 10 & 15 \end{bmatrix};$$

for the horizontal direction.

A value $\cdot$ indicates that the index is out of global bounds. Practically, this is checked by seeing whether the total index does not exceed the global bounds.

Returns

ierr

Parameters

`in,out`	*vac*	vacuum variables
`in`	*style*	style of vacuum (1: field-line 3-D, 2: axisymmetric)
`in`	*n_bnd*	total number of points in boundary
`in`	*prim↩ _x*	primary mode number

in	*n_ang*	number of angles (1) and number of field lines (2)
in	*jq*	iota (tor. flux) or q (pol. flux) at edge

Definition at line 120 of file vac_vars.f90.

Here is the call graph for this function:

B.42.2.5 set_loc_lims()

```
subroutine, public vac_vars::set_loc_lims (
            integer, intent(in) n_loc,
            integer, intent(in) bs,
            integer, intent(in) ind_p,
            integer, intent(in) n_p,
            integer, dimension(:,:), intent(inout), allocatable lims)
```

Calculates the limits in local index.

See also

See init_var() for an explanation.

Parameters

in	*n_loc*	number of points owned by local process
in	*bs*	block size
in	*ind_p*	index of current process in this dimension
in	*n_p*	number of processes in this dimension
in,out	*lims*	limits for different subregions in this dimension

Definition at line 260 of file vac_vars.f90.

Here is the caller graph for this function:

B.42.3 Variable Documentation

B.42.3.1 n_alloc_vacs

```
integer, public vac_vars::n_alloc_vacs
```

nr. of allocated vacs

Note

Debug version only

Definition at line 23 of file vac_vars.f90.

B.43 vmec_ops Module Reference

Operations that concern the output of VMEC.

Functions/Subroutines

- integer function, public read_vmec (n_r_in, use_pol_flux_v)

  Reads the VMEC equilibrium data.
- subroutine, public normalize_vmec

  Normalizes VMEC input.

B.43.1 Detailed Description

Operations that concern the output of VMEC.

B.43.2 Function/Subroutine Documentation

B.43.2.1 normalize_vmec()

```
subroutine, public vmec_ops::normalize_vmec
```

Normalizes VMEC input.

Note

The normal VMEC coordinate runs from 0 to 1, whatever the normalization.

Definition at line 327 of file VMEC_ops.f90.

Here is the caller graph for this function:

B.43.2.2 read_vmec()

```
integer function, public vmec_ops::read_vmec (
            integer, intent(inout) n_r_in,
            logical, intent(inout) use_pol_flux_v)
```

Reads the VMEC equilibrium data.

Returns

ierr

Parameters

in,out	*n_r_in*	nr. of normal points in input grid
in,out	*use_pol_flux↩ _v*	.true. if VMEC equilibrium is based on poloidal flux

Definition at line 33 of file VMEC_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.44 vmec_utilities Module Reference

Numerical utilities related to the output of VMEC.

Interfaces and Types

- interface fourier2real

  Inverse Fourier transformation, from VMEC.

Functions/Subroutines

- integer function, public calc_trigon_factors (theta, zeta, trigon_factors)

  Calculate the trigonometric cosine and sine factors.

Variables

- logical, public debug_calc_trigon_factors = .false.

  plot debug information for calc_trigon_factors()

B.44.1 Detailed Description

Numerical utilities related to the output of VMEC.

B.44.2 Function/Subroutine Documentation

B.44.2.1 calc_trigon_factors()

```
integer function, public vmec_utilities::calc_trigon_factors (
                real(dp), dimension(:,:,:), intent(in) theta,
                real(dp), dimension(:,:,:), intent(in) zeta,
                real(dp), dimension(:,:,:,:,:), intent(inout), allocatable trigon_factors)
```

Calculate the trigonometric cosine and sine factors.

This is done on a grid (`1:mnmax_V`) at given 3D arrays for the (VMEC) E(quilibrium) angles θ_E and ζ_E.

The dimensions of the output array are

`(1:mnmax_V,1:n_ang_1,1:n_ang_2,1:n_r,1:2)`

where `mnmax_V` is the number of modes in VMEC `n_r` is the total number of normal points.

See also

See grid_vars.grid_type for a discussion on `ang_1` and `ang_2`.

Returns

ierr

Parameters

`in`	*theta*	poloidal angles in equilibrium coords.
`in`	*zeta*	toroidal angles in equilibrium coords.
`in,out`	*trigon_factors*	trigonometric factor cosine and sine at these angles

Definition at line 273 of file VMEC_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.44.3 Variable Documentation

B.44.3.1 debug_calc_trigon_factors

```
logical, public vmec_utilities::debug_calc_trigon_factors = .false.
```

plot debug information for calc_trigon_factors()

Note

Debug version only

Definition at line 22 of file VMEC_utilities.f90.

B.45 vmec_vars Module Reference

Variables that concern the output of VMEC.

Functions/Subroutines

- subroutine, public dealloc_vmec ()

  Deallocates VMEC quantities that are not used anymore.

Variables

- integer, dimension(:,:), allocatable, public mn_v
  m and n of modes
- real(dp), public b_0_v
  the magnitude of B at the magnetic axis, $\theta = \zeta = 0$
- real(dp), dimension(:,:), allocatable, public flux_t_v
  toroidal flux
- real(dp), dimension(:,:), allocatable, public flux_p_v
  poloidal flux
- real(dp), dimension(:,:), allocatable, public pres_v
  pressure
- real(dp), dimension(:,:), allocatable, public rot_t_v
  rotational transform
- real(dp), dimension(:,:), allocatable, public q_saf_v
  safety factor
- real(dp), dimension(:,:,:), allocatable, public r_v_c
  Coeff. of R in sine series (FM) and norm. deriv.
- real(dp), dimension(:,:,:), allocatable, public r_v_s
  Coeff. of R in cosine series (FM) and norm. deriv.
- real(dp), dimension(:,:,:), allocatable, public z_v_c
  Coeff. of Z in sine series (FM) and norm. deriv.
- real(dp), dimension(:,:,:), allocatable, public z_v_s
  Coeff. of Z in cosine series (FM) and norm. deriv.
- real(dp), dimension(:,:,:), allocatable, public l_v_c
  Coeff. of λ in sine series (HM) and norm. deriv.
- real(dp), dimension(:,:,:), allocatable, public l_v_s
  Coeff. of λ in cosine series (HM) and norm. deriv.
- real(dp), dimension(:,:,:), allocatable, public jac_v_c
  Coeff. of $\mathcal{J}$ in sine series (HM and FM) and norm. deriv.
- real(dp), dimension(:,:,:), allocatable, public jac_v_s
  Coeff. of $\mathcal{J}$ in cosine series (HM and FM) and norm. deriv.
- real(dp), dimension(:,:,:), allocatable, public b_v_sub_c
  Coeff. of B_i in sine series (r,theta,phi) (FM).
- real(dp), dimension(:,:,:), allocatable, public b_v_sub_s
  Coeff. of B_i in cosine series (r,theta,phi) (FM).
- real(dp), dimension(:,:), allocatable, public b_v_c
  Coeff. of magnitude of B in sine series (HM and FM).
- real(dp), dimension(:,:), allocatable, public b_v_s
  Coeff. of magnitude of B in cosine series (HM and FM).
- real(dp), dimension(:,:), allocatable, public j_v_sup_int
  Integrated poloidal and toroidal current (FM).

B.45.1 Detailed Description

Variables that concern the output of VMEC.

B.45.2 Function/Subroutine Documentation

B.45.2.1 dealloc_vmec()

```
subroutine, public vmec_vars::dealloc_vmec
```

Deallocates VMEC quantities that are not used anymore.

Definition at line 56 of file VMEC_vars.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.45.3 Variable Documentation

B.45.3.1 b_0_v

```
real(dp), public vmec_vars::b_0_v
```

the magnitude of B at the magnetic axis, $\theta = \zeta = 0$

Definition at line 33 of file VMEC_vars.f90.

B.45.3.2 b_v_c

```
real(dp), dimension(:,:), allocatable, public vmec_vars::b_v_c
```

Coeff. of magnitude of B in sine series (HM and FM).

Note

Debug version only

Definition at line 50 of file VMEC_vars.f90.

B.45.3.3 b_v_s

```
real(dp), dimension(:,:), allocatable, public vmec_vars::b_v_s
```

Coeff. of magnitude of B in cosine series (HM and FM).

Note

Debug version only

Definition at line 51 of file VMEC_vars.f90.

B.45.3.4 b_v_sub_c

```
real(dp), dimension(:,:,:), allocatable, public vmec_vars::b_v_sub_c
```

Coeff. of B_i in sine series (r,theta,phi) (FM).

Note

Debug version only

Definition at line 47 of file VMEC_vars.f90.

B.45.3.5 b_v_sub_s

```
real(dp), dimension(:,:,:), allocatable, public vmec_vars::b_v_sub_s
```

Coeff. of B_i in cosine series (r,theta,phi) (FM).

Note

Debug version only

Definition at line 48 of file VMEC_vars.f90.

B.45.3.6 flux_p_v

```
real(dp), dimension(:,:), allocatable, public vmec_vars::flux_p_v
```

poloidal flux

Definition at line 35 of file VMEC_vars.f90.

B.45.3.7 flux_t_v

```
real(dp), dimension(:,:), allocatable, public vmec_vars::flux_t_v
```

toroidal flux

Definition at line 34 of file VMEC_vars.f90.

B.45.3.8 j_v_sup_int

```
real(dp), dimension(:,:), allocatable, public vmec_vars::j_v_sup_int
```

Integrated poloidal and toroidal current (FM).

Note

Debug version only

Definition at line 52 of file VMEC_vars.f90.

B.45.3.9 jac_v_c

```
real(dp), dimension(:,:,:), allocatable, public vmec_vars::jac_v_c
```

Coeff. of $\mathcal{J}$ in sine series (HM and FM) and norm. deriv.

Definition at line 45 of file VMEC_vars.f90.

B.45.3.10 jac_v_s

```
real(dp), dimension(:,:,:), allocatable, public vmec_vars::jac_v_s
```

Coeff. of $\mathcal{J}$ in cosine series (HM and FM) and norm. deriv.

Definition at line 46 of file VMEC_vars.f90.

B.45.3.11 l_v_c

```
real(dp), dimension(:,:,:), allocatable, public vmec_vars::l_v_c
```

Coeff. of λ in sine series (HM) and norm. deriv.

Definition at line 43 of file VMEC_vars.f90.

B.45.3.12 l_v_s

```
real(dp), dimension(:,:,:), allocatable, public vmec_vars::l_v_s
```

Coeff. of λ in cosine series (HM) and norm. deriv.

Definition at line 44 of file VMEC_vars.f90.

B.45.3.13 mn_v

```
integer, dimension(:,:), allocatable, public vmec_vars::mn_v
```

m and n of modes

Definition at line 32 of file VMEC_vars.f90.

B.45.3.14 pres_v

```
real(dp), dimension(:,:), allocatable, public vmec_vars::pres_v
```

pressure

Definition at line 36 of file VMEC_vars.f90.

B.45.3.15 q_saf_v

```
real(dp), dimension(:,:), allocatable, public vmec_vars::q_saf_v
```

safety factor

Definition at line 38 of file VMEC_vars.f90.

B.45.3.16 r_v_c

```
real(dp), dimension(:,:,:), allocatable, public vmec_vars::r_v_c
```

Coeff. of R in sine series (FM) and norm. deriv.

Definition at line 39 of file VMEC_vars.f90.

B.45.3.17 r_v_s

```
real(dp), dimension(:,:,:), allocatable, public vmec_vars::r_v_s
```

Coeff. of R in cosine series (FM) and norm. deriv.

Definition at line 40 of file VMEC_vars.f90.

B.45.3.18 rot_t_v

```
real(dp), dimension(:,:), allocatable, public vmec_vars::rot_t_v
```

rotational transform

Definition at line 37 of file VMEC_vars.f90.

B.45.3.19 z_v_c

```
real(dp), dimension(:,:,:), allocatable, public vmec_vars::z_v_c
```

Coeff. of Z in sine series (FM) and norm. deriv.

Definition at line 41 of file VMEC_vars.f90.

B.45.3.20 z_v_s

```
real(dp), dimension(:,:,:), allocatable, public vmec_vars::z_v_s
```

Coeff. of Z in cosine series (FM) and norm. deriv.

Definition at line 42 of file VMEC_vars.f90.

B.46 x_ops Module Reference

Operations considering perturbation quantities.

Interfaces and Types

- interface calc_x
  Calculates either vectorial or tensorial perturbation variables.
- interface redistribute_output_x
  Redistribute the perturbation variables.
- interface print_output_x
  Print either vectorial or tensorial perturbation quantities of a certain order to an output file.

Functions/Subroutines

- integer function, public init_modes (grid_eq, eq)

  Initializes some variables concerning the mode numbers.
- integer function, public setup_modes (mds, grid_eq, grid, plot_name)

  Sets up some variables concerning the mode numbers.
- integer function, public check_x_modes (grid_eq, eq)

  Checks whether the high-n approximation is valid:
- integer function, public calc_res_surf (mds, grid_eq, eq, res_surf, info, jq)

  Calculates resonating flux surfaces for the perturbation modes.
- integer function, public resonance_plot (mds, grid_eq, eq)

  plot q-profile or ι-profile in flux coordinates with $nq - m = 0$ or $n - \iota m = 0$ indicated if requested.
- integer function, public calc_magn_ints (grid_eq, grid_x, eq, x, x_int, prev_style, lim_sec_x)

  Calculate the magnetic integrals from $\widetilde{PV}^i_{k,m}$ and $\widetilde{KV}^i_{k,m}$ in an equidistant grid where the step size can vary depending on the normal coordinate.
- integer function, public divide_x_jobs (arr_size)

  Divides the perturbation jobs.
- integer function, public print_debug_x_1 (mds, grid_x, x_1)

  Prints debug information for X_1 driver.
- integer function, public print_debug_x_2 (mds, grid_x, x_2_int)

  Prints debug information for X_2 driver.

Variables

- logical, public debug_check_x_modes_2 = .false.

  plot debug information for check_x_modes_2()

B.46.1 Detailed Description

Operations considering perturbation quantities.

B.46.2 Function/Subroutine Documentation

B.46.2.1 calc_magn_ints()

```
integer function, public x_ops::calc_magn_ints (
                type(grid_type), intent(in) grid_eq,
                type(grid_type), intent(in) grid_x,
                type(eq_2_type), intent(in), target eq,
                type(x_2_type), intent(in) x,
                type(x_2_type), intent(inout) x_int,
                integer, intent(in), optional prev_style,
                integer, dimension(2,2), intent(in), optional lim_sec_x)
```

Calculate the magnetic integrals from $\widetilde{PV}^i_{k,m}$ and $\widetilde{KV}^i_{k,m}$ in an equidistant grid where the step size can vary depending on the normal coordinate.

All the variables should thus be field-line oriented. The result is saved in the first index of the X variables, the other can be ignored.

Using `prev_style`, the results of this calculation on X can be combined with the ones already present in `X_int`.

- `prev_style=0`: Overwrite [def].
- `prev_style=1`: Add to current integral.
- `prev_style=2`: Divide by 2 and add to current integral. Also modify the indices of the current integral:
  - for `magn_int_style` = 1: `1 2 2 2 2 2 1 -> 2 2 2 2 2 2`,
  - for `magn_int_style` = 2: `1 3 3 2 3 3 1 -> 3 2 3 3 2 3`.
- `prev_style=3`: Add to current integral. Also modify the indices of the current integral as in `prev_style` = 2.
- else: ignore previous magnetic integrals.

Therefore, it is to be used in order. For example:

1. (R=1,E=1): style 0,
2. → (R=1,E=2): style 1,
3. → (R=2,E=1): style 2,
4. → (R=2,E=2): style 3.

Afterwards, it would cycle through 2 and 3, as style 0 and 1 are only for the first Richardson level.

Note

1. The variable type for the integrated tensorial perturbation variables is also `X_2`, but it is assumed that the first index has dimension 1, the rest is ignored.
2. Simpson's 3/8 rule converges faster than the trapezoidal rule, but normally needs a better starting point (i.e. higher `min_n_par_X`)
3. For alpha style 1, Weyl's lemma is used to convert the surface integral into a line integral along the magnetic field. This means that the resulting line integral still needs to be multiplied by $\frac{2\pi}{M}$, where M is the number of times the poloidal or toroidal angle completes a full 2π tour, depending on `use_pol_flux_F` [6].
4. For alpha style 2, this factor is just the step size in alpha.

See also

See x_vars.x_2_type.

Returns

ierr

Parameters

`in`	*grid_eq*	equilibrium grid

in	*grid_x*	perturbation grid
in	*eq*	metric equilibrium
in	*x*	tensorial perturbation variables
in,out	*x_int*	interpolated tensorial perturbation variables, containing all mode combinations
in	*prev_style*	style to treat X_prev
in	*lim_sec_x*	limits of m_X (pol flux) or n_X (tor flux) for both dimensions

Definition at line 3079 of file X_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.46.2.2 calc_res_surf()

```
integer function, public x_ops::calc_res_surf (
                type(modes_type), intent(in) mds,
                type(grid_type), intent(in) grid_eq,
                type(eq_1_type), intent(in) eq,
                real(dp), dimension(:,:), intent(inout), allocatable res_surf,
                logical, intent(in), optional info,
                real(dp), dimension(:), intent(inout), optional, allocatable jq)
```

Calculates resonating flux surfaces for the perturbation modes.

The output consists of mode number, resonating normal position and the fraction $\frac{m}{n}$ or $\frac{n}{m}$. for those modes for which a solution is found that is within the plasma range.

It contains three pieces of information:

- `(:,1)`: the mode index
- `(:,2)`: the radial position in Flux coordinates
- `(:,3)`: the fraction $\frac{m}{n}$ or $\frac{n}{m}$

for every single mode in `sec` of `mds`, which can be tabulated in an arbitrary grid, not necessarily the equilibrium one.

Optionally, the total safety factor or rotational transform can be returned to the master.

Also, information can be displayed with info.

Returns

ierr

Parameters

`in`	*mds*	general modes variables
`in`	*grid_eq*	equilibrium grid_eq
`in`	*eq*	flux equilibrium
`in,out`	*res_surf*	resonant surface
`in`	*info*	if info is displayed
`in,out`	*jq*	either safety factor or rotational transform

Definition at line 1637 of file X_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.46.2.3 check_x_modes()

```
integer function, public x_ops::check_x_modes (
                type(grid_type), intent(in) grid_eq,
                type(eq_1_type), intent(in) eq)
```

Checks whether the high-n approximation is valid:

This depends on the `X_style`:

For X_style 1 (prescribed): every mode should resonate at least somewhere in the whole normal range:

- $\frac{|nq-m|}{|n|} < T$ and $\frac{|nq-m|}{|m|} < T$ for poloidal flux
- $\frac{|q-\iota m|}{|m|} < T$ and $\frac{|n-\iota m|}{|n|} < T$ for toroidal flux

where T = `tol` $<<$ 1.

This condition is determined by the sign of q (or ι) and given by:

- max $\left(q_{\text{min}} - T, \frac{q_{\text{min}}}{1+T}\right) < \frac{m}{n} <$ min $\left(q_{\text{max}} + T, \frac{q_{\text{max}}}{1-T}\right), q > 0$
- max $\left(q_{\text{min}} - T, \frac{q_{\text{min}}}{1-T}\right) < \frac{m}{n} <$ min $\left(q_{\text{max}} + T, \frac{q_{\text{max}}}{1+T}\right), q < 0$

for poloidal flux.

For toroidal flux, q should be replaced by ι and $\frac{m}{n}$ by $\frac{n}{m}$.

For `X_style` 2 (fast): the resonance has been taken care of, but it remains to be checked whether the number of modes is efficient.

Returns

ierr

Parameters

in	*grid_eq*	equilibrium grid
in	*eq*	flux equilibrium

Definition at line 1349 of file X_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.46.2.4 divide_x_jobs()

```
integer function, public x_ops::divide_x_jobs (
                integer, intent(in) arr_size)
```

Divides the perturbation jobs.

This concerns the calculation of the magnetic integrals of blocks of tensorial perturbation variables. These are set up using the equilibrium and vectorial perturbation variables that are stored in memory, but only the integrated tensorial result is stored in memory, with negligible variables size.

The size of the (k,m) pairs to be calculated is determined by looking at what fits in memory when the equilibrium and vectorial perturbation variables are stored in memory first.

Returns

ierr

Parameters

in	*arr_size*	array size (using loc_n_r)

Definition at line 3363 of file X_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.46.2.5 init_modes()

```
integer function, public x_ops::init_modes (
                type(grid_type), intent(in) grid_eq,
                type(eq_1_type), intent(in) eq)
```

Initializes some variables concerning the mode numbers.

Setup minimum and maximum of mode numbers at every flux surface in equilibrium coordinates

- `min_n_X`, `max_n_X`
- `min_m_X`, `max_m_X`,

It functions depending on the `X_style` used: 1 (prescribed) or 2 (fast). For the fast style, at every flux surface the range of modes is sought that resonates most:

- $m = nq \pm \frac{N}{2}$ for poloidal flux
- $n = \iota m \pm \frac{N}{2}$ for toroidal flux

where N = `n_mod_X`.

However, at the same time, both `m` and `n` have to be larger, in absolute value, than `min_sec_X`. Therefore, the range of width `n_mod_X` can be shifted upwards.

Returns

ierr

Parameters

in	*grid_eq*	equilibrium grid

in	*eq*	flux equilibrium

Definition at line 918 of file X_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.46.2.6 print_debug_x_1()

```
integer function, public x_ops::print_debug_x_1 (
                type(modes_type), intent(in) mds,
                type(grid_type), intent(in) grid_x,
                type(x_1_type), intent(in) x_1)
```

Prints debug information for X_1 driver.

Parameters

in	*mds*	general modes variables
in	*grid↩ _x*	perturbation grid
in	*x_1*	vectorial X variables

Definition at line 3489 of file X_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.46.2.7 print_debug_x_2()

```
integer function, public x_ops::print_debug_x_2 (
                type(modes_type), intent(in) mds,
                type(grid_type), intent(in) grid_x,
                type(x_2_type), intent(in) x_2_int)
```

Prints debug information for X_2 driver.

Parameters

in	*mds*	general modes variables
in	*grid_x*	perturbation grid
in	*x_2_int*	tensorial X variables

Definition at line 3632 of file X_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.46.2.8 resonance_plot()

```
integer function, public x_ops::resonance_plot (
            type(modes_type), intent(in) mds,
            type(grid_type), intent(in) grid_eq,
            type(eq_1_type), intent(in) eq)
```

plot q-profile or ι-profile in flux coordinates with $nq - m = 0$ or $n - \iota m = 0$ indicated if requested.

The plot will be done in the grid in which `mds` is tabulated, which is not necessarily the equilibrium one.

Returns

ierr

Parameters

in	*mds*	general modes variables
in	*grid_eq*	equilibrium grid
in	*eq*	flux equilibrium

Definition at line 1813 of file X_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.46.2.9 setup_modes()

```
integer function, public x_ops::setup_modes (
            type(modes_type), intent(inout), target mds,
            type(grid_type), intent(in) grid_eq,
            type(grid_type), intent(in) grid,
            character(len=*), intent(in), optional plot_name)
```

Sets up some variables concerning the mode numbers.

Apart from the mode numbers

- n, m,

also the variables of the secondary modes

- sec,

are set up in the coordinates of the grid passed. The normal values of this grid are saved as well, in

- r_F.

All variables are tabulated in the full normal grid. The mode indices, however, are chosen so that there is maximum overlap between different normal positions. This is trivial for `X_style` 1 (prescribed), but for `X_style` 2 (fast), this is best explained by an example:

B.46.2.10 kd m1 m2 m3

1 10 11 12 <- start 2 10 11 12 <- no change in limits 2 13 11 12 <- limits shift up by one 3 13 11 12 <- no change in limits 4 13 14 12 <- limits shift up by one 5 16 14 15 <- limits shift up by two 6 13 14 15 <- limits shift down by one

This procedure makes use of the global variables

- `min_m_X`, max_m_X, min_n_X, max_m_X

that have to be set up using init_nm_x().

Optionally, `n` and `m` can be plot.

Returns

ierr

Parameters

`in,out`	*mds*	modes variables

in	*grid_eq*	equilibrium grid
in	*grid*	grid at which to calculate modes
in	*plot_name*	name to be used when plotting n and m

Definition at line 1059 of file X_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.46.3 Variable Documentation

B.46.3.1 debug_check_x_modes_2

```
logical, public x_ops::debug_check_x_modes_2 = .false.
```

plot debug information for check_x_modes_2()

Note

Debug version only

Definition at line 26 of file X_ops.f90.

B.47 x_utilities Module Reference

Numerical utilities related to perturbation operations.

Interfaces and Types

- interface sec_ind_loc2tot

  Returns the `sec_ind_tot` used to refer to a perturbation quantity.

Functions/Subroutines

- subroutine, public get_sec_x_range (sec_x_range_loc, sec_x_range_tot, m, sym, lim_sec_x)

  Gets one of the the local ranges of contiguous tensorial perturbation variables to be printed or read during one call of the corresponding HDF5 variables.
- logical function, public do_x ()

  Tests whether this perturbatino job should be done.
- logical function, public is_necessary_x (sym, sec_x_id, lim_sec_x)

  Determines whether a variable needs to be considered:
- integer function, public calc_memory_x (ord, arr_size, n_mod, mem_size)

  Calculate memory in MB necessary for X variables.
- integer function, public trim_modes (mds_i, mds_o, id_lim_i, id_lim_o)

  Limit input mode range to output mode range.

B.47.1 Detailed Description

Numerical utilities related to perturbation operations.

B.47.2 Function/Subroutine Documentation

B.47.2.1 calc_memory_x()

```
integer function, public x_utilities::calc_memory_x (
               integer, intent(in) ord,
               integer, intent(in) arr_size,
               integer, intent(in) n_mod,
               real(dp), intent(inout) mem_size)
```

Calculate memory in MB necessary for X variables.

This depends on the order:

- order 1:
  - `4x n_par_X x n_geo x loc_n_r x n_mod`
- order 2:
  - `2x n_par_X x n_geo x loc_n_r x n_mod^2`
  - `4x n_par_X x n_geo x loc_n_r x n_mod(n_mod+1)/2`
- higher order: not used

where `n_par_X x n_geo x loc_n_r` should be passed as `arr_size` and `n_mod` as well.

Returns

ierr

Parameters

in	*ord*	order of data

in	*arr_size*	size of part of X array
in	*n_mod*	number of modes
in,out	*mem_size*	total size

Definition at line 235 of file X_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.47.2.2 do_x()

`logical function, public x_utilities::do_x`

Tests whether this perturbatino job should be done.

Also increments `X_job_nr`

Definition at line 178 of file X_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.47.2.3 get_sec_x_range()

```
subroutine, public x_utilities::get_sec_x_range (
                integer, dimension(2), intent(inout) sec_x_range_loc,
                integer, dimension(2), intent(inout) sec_x_range_tot,
                integer, intent(in) m,
                logical, intent(in) sym,
                integer, dimension(2,2), intent(in), optional lim_sec_x)
```

Gets one of the the local ranges of contiguous tensorial perturbation variables to be printed or read during one call of the corresponding HDF5 variables.

More specifically, a range of indices `k` in the first dimension is given for every value of the indices `m` in the second dimension.

An example is now given for the subrange `[2:3,2:5]` of a the total range `[1:5, 1:5]`. For asymmetric variables the situation is simple: The `k` range is `[2:3]` for all 5 values of `m`. However, for symmetric variables, the upper diagonal values are not stored, which gives `k` ranges `[2:3]`, `[3:3]` and no range for `m` = 4 and 5.

This routine then translates these ranges to the corresponding 1-D ranges that are used in the actual variables. For above example, the total indices are

$$\begin{pmatrix} 1 & 6 & 11 & 16 & 21 \\ 2 & 7 & 12 & 17 & 22 \\ 3 & 8 & 13 & 18 & 23 \\ 4 & 9 & 14 & 19 & 24 \\ 5 & 10 & 15 & 20 & 25 \end{pmatrix} \rightarrow [7:8], [12:13], [17:18] \text{ and } [22:23],$$

for asymmetric variables and

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 6 & 7 & 8 & 9 \\ 3 & 7 & 10 & 11 & 12 \\ 4 & 8 & 11 & 13 & 14 \\ 5 & 9 & 12 & 14 & 15 \end{pmatrix} \rightarrow [6:7], [10:10], [:] \text{ and } [:],$$

for symmetric variables.

These can then related to the local indices for the variables in this perturbation job.

For above example, the results are:

`[1:2], [3:4], [5:6] and [7:8],`

for asymmetric variables and

`[1:2], [3:3], [:] and [:],`

for symmetric variables.

As can be seen, the local ranges of the variables in the submatrix of this perturbation job are (designed to be) contiguous, but the total ranges of the variables in the submatrix are clearly not in general.

The procedure outputs both the local and total ranges.

Parameters

`in,out`	*sec_x_range_loc*	start and end of local range in dimension 1 (vertical)

in,out	*sec_x_range_tot*	start and end of total range in dimension 1 (vertical)
in	*m*	dimension 2 (horizontal)
in	*sym*	whether the variable is symmetric
in	*lim_sec_x*	limits of m_X (pol. flux) or n_X (tor. flux)

Definition at line 127 of file X_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

B.47.2.4 is_necessary_x()

```
logical function, public x_utilities::is_necessary_x (
                logical, intent(in) sym,
                integer, dimension(2), intent(in) sec_x_id,
                integer, dimension(2,2), intent(in), optional lim_sec_x)
```

Determines whether a variable needs to be considered:

This depends on whether the quantity is symmetric or not:

- Only if it is on or below the diagonal for symmetric quantities.
- Always for asymmetric quantities

Parameters

in	*sym*	whether the variable is symmetric

in	*sec_x_id*	mode indices
in	*lim_sec↩ _x*	limits of m_X (pol flux) or n_X (tor flux) for both dimensions

Definition at line 196 of file X_utilities.f90.

Here is the caller graph for this function:

B.47.2.5 trim_modes()

```
integer function, public x_utilities::trim_modes (
                type(modes_type), intent(in) mds_i,
                type(modes_type), intent(in) mds_o,
                integer, dimension(2), intent(inout) id_lim_i,
                integer, dimension(2), intent(inout) id_lim_o)
```

Limit input mode range to output mode range.

Parameters

in	*mds_i*	general modes variables for input
in	*mds_o*	general modes variables for output
in,out	*id_lim↩ _i*	limits on input modes
in,out	*id_lim↩ _o*	limits on output modes

Definition at line 292 of file X_utilities.f90.

Here is the caller graph for this function:

B.48 x_vars Module Reference

Variables pertaining to the perturbation quantities.

Interfaces and Types

- type modes_type
  mode number type
- type x_1_type
  vectorial perturbation type
- type x_2_type
  tensorial perturbation type
- interface set_nm_x
  Sets n_X and m_X.

Functions/Subroutines

- subroutine init_x_1 (x, mds, grid_x, lim_sec_x)
  Initializes a vectorial perturbation.
- subroutine init_x_2 (x, mds, grid_x, lim_sec_x, is_field_averaged)
  Initializes a tensorial perturbation.
- subroutine copy_x_1 (x_i, mds, grid_i, x_o)
  Deep copy of vectorial perturbation variables.
- subroutine copy_x_2 (x_i, mds, grid_i, x_o)
  Deep copy of tensorial perturbation variables.
- integer function, public set_nn_mod (sym, lim_sec_x)
  Sets number of entries for tensorial perturbation variables.
- subroutine dealloc_mds (mds)
  Deallocates modes variables.
- subroutine dealloc_x_1 (x)
  Deallocates vectorial perturbation variables.
- subroutine dealloc_x_2 (x)
  Deallocates tensorial perturbation variables.

Variables

- type(modes_type), public mds_x
  modes variables for perturbation grid
- type(modes_type), public mds_sol
  modes variables for solution grid
- integer, public prim_x
  n_X (pol. flux) or m_X (tor. flux)
- integer, public min_sec_x
  m_X (pol. flux) or n_X (tor. flux) (only for X style 1)
- integer, public max_sec_x
  m_X (pol. flux) or n_X (tor. flux) (only for\ c X style 1)
- integer, public n_mod_x

size of m_X (pol. flux) or n_X (tor. flux)

- integer, public min_nm_x = 5

  minimum for the high-n theory (debable)
- integer, dimension(:), allocatable, public min_n_x

  lowest poloidal mode number m_X, in total eq grid
- integer, dimension(:), allocatable, public max_n_x

  highest poloidal mode number m_X, in total eq grid
- integer, dimension(:), allocatable, public min_m_x

  lowest poloidal mode number m_X, in total eq grid
- integer, dimension(:), allocatable, public max_m_x

  highest poloidal mode number m_X, in total eq grid
- real(dp), public min_r_sol

  min. normal range for pert.
- real(dp), public max_r_sol

  max. normal range for pert.
- integer, public n_alloc_x_1s

  nr. of allocated X_1's
- integer, public n_alloc_x_2s

  nr. of allocated X_2's

B.48.1 Detailed Description

Variables pertaining to the perturbation quantities.

B.48.2 Function/Subroutine Documentation

B.48.2.1 copy_x_1()

```
subroutine x_vars::copy_x_1 (
            class(x_1_type), intent(in) x_i,
            type(modes_type), intent(in) mds,
            type(grid_type), intent(in) grid_i,
            type(x_1_type), intent(inout) x_o)
```

Deep copy of vectorial perturbation variables.

Parameters

in	*x_i*	X_1 to be copied
in	*mds*	general modes variables
in	*grid↩ _i*	grid of eq_i
in,out	*x_o*	copied X_1

Definition at line 337 of file X_vars.f90.

B.48.2.2 copy_x_2()

```
subroutine x_vars::copy_x_2 (
                class(x_2_type), intent(in) x_i,
                type(modes_type), intent(in) mds,
                type(grid_type), intent(in) grid_i,
                type(x_2_type), intent(inout) x_o)
```

Deep copy of tensorial perturbation variables.

Parameters

in	*x_i*	X_2 to be copied
in	*mds*	general modes variables
in	*grid↩ _i*	grid of eq_i
in,out	*x_o*	copied X_1

Definition at line 358 of file X_vars.f90.

B.48.2.3 dealloc_mds()

```
subroutine x_vars::dealloc_mds (
                class(modes_type), intent(inout) mds)
```

Deallocates modes variables.

Parameters

in,out	*mds*	modes variables to be deallocated

Definition at line 411 of file X_vars.f90.

B.48.2.4 dealloc_x_1()

```
subroutine x_vars::dealloc_x_1 (
                class(x_1_type), intent(inout) x)
```

Deallocates vectorial perturbation variables.

Parameters

in,out	*x*	perturbation variables to be deallocated

Definition at line 427 of file X_vars.f90.

Here is the call graph for this function:

B.48.2.5 dealloc_x_2()

```
subroutine x_vars::dealloc_x_2 (
            class(x_2_type), intent(inout) x)
```

Deallocates tensorial perturbation variables.

Parameters

in,out	x	perturbation variables to be deallocated

Definition at line 473 of file X_vars.f90.

Here is the call graph for this function:

B.48.2.6 init_x_1()

```
subroutine x_vars::init_x_1 (
            class(x_1_type), intent(inout) x,
            type(modes_type), intent(in) mds,
            type(grid_type), intent(in) grid_x,
            integer, dimension(2), intent(in), optional lim_sec_x)
```

Initializes a vectorial perturbation.

Allocates the variables, the number of modes, as well as `n` and `m`.

Optionally, the secondary mode numbers can be specified (`m` if poloidal flux is used and `n` if toroidal flux). By default, they are taken from the global `X_vars` variables.

Note

If the lowest limits of the grid is not 1 (e.g. `grid_soli_min = 1` for first process), the input variable `i_min` should be set to set correctly. For a full grid, it should be set to 1.

Parameters

`in,out`	*x*	vectorial perturbation variables

in	*mds*	general modes variables
in	*grid_x*	perturbation grid
in	*lim_sec↩ _x*	limits of m_X (pol. flux) or n_X (tor. flux)

Definition at line 203 of file X_vars.f90.

Here is the call graph for this function:

B.48.2.7 init_x_2()

```
subroutine x_vars::init_x_2 (
            class(x_2_type), intent(inout) x,
            type(modes_type), intent(in) mds,
            type(grid_type), intent(in) grid_x,
            integer, dimension(2,2), intent(in), optional lim_sec_x,
            logical, intent(in), optional is_field_averaged)
```

Initializes a tensorial perturbation.

See also

See init_X_1().

Note

The tensorial perturbation type can also be used for field-aligned variables, in which case the first index is assumed to have dimension 1 only. This can be triggered using `is_field_averaged`. There is no difference between a tensorial perturbation type with size of first dimension set to one through the use of this flag, or through other means.

Parameters

in,out	*x*	tensorial perturbation variables
in	*mds*	general modes variables

in	*grid_x*	perturbation grid
in	*lim_sec_x*	limits of m_X (pol. flux) or n_X (tor. flux) for both dimensions
in	*is_field_averaged*	if field-aligned, only one dimension for first index

Definition at line 270 of file X_vars.f90.

Here is the call graph for this function:

B.48.2.8 set_nn_mod()

```
integer function, public x_vars::set_nn_mod (
                logical, intent(in) sym,
                integer, dimension(2,2), intent(in), optional lim_sec_x)
```

Sets number of entries for tensorial perturbation variables.

Parameters

in	*sym*	whether the variable is symmetric
in	*lim_sec_x*	limits of m_X (pol flux) or n_X (tor flux) for both dimensions

Definition at line 382 of file X_vars.f90.

Here is the caller graph for this function:

B.48.3 Variable Documentation

B.48.3.1 max_m_x

```
integer, dimension(:), allocatable, public x_vars::max_m_x
```

highest poloidal mode number `m_X`, in total eq grid

Definition at line 134 of file X_vars.f90.

B.48.3.2 max_n_x

```
integer, dimension(:), allocatable, public x_vars::max_n_x
```

highest poloidal mode number `m_X`, in total eq grid

Definition at line 132 of file X_vars.f90.

B.48.3.3 max_r_sol

```
real(dp), public x_vars::max_r_sol
```

max. normal range for pert.

Definition at line 136 of file X_vars.f90.

B.48.3.4 max_sec_x

```
integer, public x_vars::max_sec_x
```

`m_X` (pol. flux) or `n_X` (tor. flux) (only for\ c X style 1)

Definition at line 128 of file X_vars.f90.

B.48.3.5 mds_sol

```
type(modes_type), public x_vars::mds_sol
```

modes variables for solution grid

Definition at line 125 of file X_vars.f90.

B.48.3.6 mds_x

```
type(modes_type), public x_vars::mds_x
```

modes variables for perturbation grid

Definition at line 124 of file X_vars.f90.

B.48.3.7 min_m_x

```
integer, dimension(:), allocatable, public x_vars::min_m_x
```

lowest poloidal mode number `m_X`, in total eq grid

Definition at line 133 of file X_vars.f90.

B.48.3.8 min_n_x

```
integer, dimension(:), allocatable, public x_vars::min_n_x
```

lowest poloidal mode number `m_X`, in total eq grid

Definition at line 131 of file X_vars.f90.

B.48.3.9 min_nm_x

```
integer, public x_vars::min_nm_x = 5
```

minimum for the high-n theory (debable)

Definition at line 130 of file X_vars.f90.

B.48.3.10 min_r_sol

```
real(dp), public x_vars::min_r_sol
```

min. normal range for pert.

Definition at line 135 of file X_vars.f90.

B.48.3.11 min_sec_x

```
integer, public x_vars::min_sec_x
```

`m_X` (pol. flux) or `n_X` (tor. flux) (only for `X` style 1)

Definition at line 127 of file X_vars.f90.

B.48.3.12 n_alloc_x_1s

```
integer, public x_vars::n_alloc_x_1s
```

nr. of allocated `X_1`'s

Note

Debug version only

Definition at line 138 of file X_vars.f90.

B.48.3.13 n_alloc_x_2s

```
integer, public x_vars::n_alloc_x_2s
```

nr. of allocated `X_2`'s

Note

Debug version only

Definition at line 139 of file X_vars.f90.

B.48.3.14 n_mod_x

```
integer, public x_vars::n_mod_x
```

size of `m_X` (pol. flux) or `n_X` (tor. flux)

Definition at line 129 of file X_vars.f90.

B.48.3.15 prim_x

```
integer, public x_vars::prim_x
```

`n_X` (pol. flux) or `m_X` (tor. flux)

Definition at line 126 of file X_vars.f90.

Appendix C

Interfaces and Types

C.1 num_utilities::add_arr_mult Interface Reference

Add to an array (3) the product of arrays (1) and (2).

Public Member Functions

- integer function add_arr_mult_3_3 (arr_1, arr_2, arr_3, deriv)
  version with `arr_1` and `arr_2` in 3 coords.
- integer function add_arr_mult_3_1 (arr_1, arr_2, arr_3, deriv)
  version with `arr_1` in 3 coords and `arr_2` only in the flux coord.
- integer function add_arr_mult_1_1 (arr_1, arr_2, arr_3, deriv)
  Version with `arr_1` and `arr_2` only in the flux coord.

C.1.1 Detailed Description

Add to an array (3) the product of arrays (1) and (2).

The derivatives are distributed between both acording to the binomial theorem. Both arrays are given on a 3-D or 1-D grid.

Returns

ierr

Definition at line 39 of file num_utilities.f90.

C.1.2 Member Function/Subroutine Documentation

C.1.2.1 add_arr_mult_1_1()

```
integer function num_utilities::add_arr_mult::add_arr_mult_1_1 (
                real(dp), dimension(1:,0:), intent(in) arr_1,
                real(dp), dimension(1:,0:), intent(in) arr_2,
                real(dp), dimension(1:), intent(out) arr_3,
                integer, dimension(3), intent(in) deriv)
```

Version with `arr_1` and `arr_2` only in the flux coord.

Parameters

in	*arr↩ _1*	arr_1

in	*arr↩ _2*	arr_2
out	*arr↩ _3*	arr_3
in	*deriv*	derivatives

Definition at line 491 of file num_utilities.f90.

C.1.2.2 add_arr_mult_3_1()

```
integer function num_utilities::add_arr_mult::add_arr_mult_3_1 (
                real(dp), dimension(1:,1:,1:,0:,0:,0:), intent(in) arr_1,
                real(dp), dimension(1:,0:), intent(in) arr_2,
                real(dp), dimension(1:,1:,1:), intent(out) arr_3,
                integer, dimension(3), intent(in) deriv)
```

version with arr_1 in 3 coords and arr_2 only in the flux coord.

Parameters

in	*arr↩ _1*	arr_1
in	*arr↩ _2*	arr_2
out	*arr↩ _3*	arr_3
in	*deriv*	derivatives

Definition at line 440 of file num_utilities.f90.

C.1.2.3 add_arr_mult_3_3()

```
integer function num_utilities::add_arr_mult::add_arr_mult_3_3 (
                real(dp), dimension(1:,1:,1:,0:,0:,0:), intent(in) arr_1,
                real(dp), dimension(1:,1:,1:,0:,0:,0:), intent(in) arr_2,
                real(dp), dimension(1:,1:,1:), intent(out) arr_3,
                integer, dimension(3), intent(in) deriv)
```

version with arr_1 and arr_2 in 3 coords.

Parameters

in	*arr↩ _1*	arr_1
in	*arr↩ _2*	arr_2

out	*arr↩ _3*	arr_3
in	*deriv*	derivatives

Definition at line 362 of file num_utilities.f90.

Here is the call graph for this function:

C.2 mpi_utilities::broadcast_var Interface Reference

Wrapper function to broadcast a single variable using MPI.

Public Member Functions

- integer function broadcast_var_real (var, source)
  real version
- integer function broadcast_var_int (var, source)
  integer version
- integer function broadcast_var_log (var, source)
  logical version
- integer function broadcast_var_complex_arr (var, source)
  complex array version
- integer function broadcast_var_real_arr (var, source)
  real array version
- integer function broadcast_var_int_arr (var, source)
  integer array version
- integer function broadcast_var_log_arr (var, source)
  logical array version

C.2.1 Detailed Description

Wrapper function to broadcast a single variable using MPI.

Returns

ierr

Definition at line 87 of file MPI_utilities.f90.

C.2.2 Member Function/Subroutine Documentation

C.2.2.1 broadcast_var_complex_arr()

```
integer function mpi_utilities::broadcast_var::broadcast_var_complex_arr (
                complex(dp), dimension(:), intent(in) var,
                integer, intent(in), optional source)
```

complex array version

Parameters

in	*var*	variable to be broadcast
in	*source*	process that sends

Definition at line 658 of file MPI_utilities.f90.

C.2.2.2 broadcast_var_int()

```
integer function mpi_utilities::broadcast_var::broadcast_var_int (
                integer, intent(in) var,
                integer, intent(in), optional source)
```

integer version

Parameters

in	*var*	variable to be broadcast
in	*source*	process that sends

Definition at line 618 of file MPI_utilities.f90.

C.2.2.3 broadcast_var_int_arr()

```
integer function mpi_utilities::broadcast_var::broadcast_var_int_arr (
                integer, dimension(:), intent(in) var,
                integer, intent(in), optional source)
```

integer array version

Parameters

in	*var*	variable to be broadcast
in	*source*	process that sends

Definition at line 700 of file MPI_utilities.f90.

C.2.2.4 broadcast_var_log()

```
integer function mpi_utilities::broadcast_var::broadcast_var_log (
               logical, intent(in) var,
               integer, intent(in), optional source)
```

logical version

Parameters

in	*var*	variable to be broadcast

in	*source*	process that sends

Definition at line 638 of file MPI_utilities.f90.

C.2.2.5 broadcast_var_log_arr()

```
integer function mpi_utilities::broadcast_var::broadcast_var_log_arr (
                logical, dimension(:), intent(in) var,
                integer, intent(in), optional source)
```

logical array version

Parameters

in	*var*	variable to be broadcast
in	*source*	process that sends

Definition at line 720 of file MPI_utilities.f90.

C.2.2.6 broadcast_var_real()

```
integer function mpi_utilities::broadcast_var::broadcast_var_real (
                real(dp), intent(in) var,
                integer, intent(in), optional source)
```

real version

Parameters

in	*var*	variable to be broadcast
in	*source*	process that sends

Definition at line 597 of file MPI_utilities.f90.

C.2.2.7 broadcast_var_real_arr()

```
integer function mpi_utilities::broadcast_var::broadcast_var_real_arr (
                real(dp), dimension(:), intent(in) var,
                integer, intent(in), optional source)
```

real array version

Parameters

in	*var*	variable to be broadcast

in	*source*	process that sends

Definition at line 679 of file MPI_utilities.f90.

C.3 num_utilities::bubble_sort Interface Reference

Sorting with the bubble sort routine.

Public Member Functions

- subroutine bubble_sort_int (a, piv)
  integer version
- subroutine bubble_sort_real (a, piv)
  real version

C.3.1 Detailed Description

Sorting with the bubble sort routine.

Optionally, the pivots can be given back.

Note

Adapted from `http://rosettacode.org/wiki/Category:Fortran`

Definition at line 237 of file num_utilities.f90.

C.3.2 Member Function/Subroutine Documentation

C.3.2.1 bubble_sort_int()

```
subroutine num_utilities::bubble_sort::bubble_sort_int (
            integer, dimension(:), intent(inout) a,
            integer, dimension(:), intent(inout), optional piv)
```

integer version

Parameters

in,out	*a*	vector to sort
in,out	*piv*	pivots

Definition at line 1538 of file num_utilities.f90.

C.3.2.2 bubble_sort_real()

```
subroutine num_utilities::bubble_sort::bubble_sort_real (
               real(dp), dimension(:), intent(inout) a,
               integer, dimension(:), intent(inout), optional piv)
```

real version

Parameters

in,out	*a*	vector to sort

in,out	*piv*	pivots

Definition at line 1504 of file num_utilities.f90.

C.4 num_utilities::calc_det Interface Reference

Calculate determinant of a matrix.

Public Member Functions

- integer function calc_det_0d (det_0d, a)
  private constant version
- integer recursive function calc_det_3d (deta, a, n)
  private array version

C.4.1 Detailed Description

Calculate determinant of a matrix.

This matrix can be defined on a 3-D grid or constant. The storage convention described in eq_vars.eq_2_type is used.

In the former case the size of the matrix (last two indices) should be small, as the direct formula employing cofactors is used through a recursive formulation.

In the latter case, lapack routines are used.

See also

Adapted from `http://dualm.wordpress.com/2012/01/06/computing-determinant-in-fortran/`

Returns

ierr

Definition at line 63 of file num_utilities.f90.

C.4.2 Member Function/Subroutine Documentation

C.4.2.1 calc_det_0d()

```
integer function num_utilities::calc_det::calc_det_0d (
            real(dp), intent(inout) det_0d,
            real(dp), dimension(:,:), intent(in) a)
```

private constant version

Parameters

in,out	*det_0d*	output

in	*a*	input

Definition at line 621 of file num_utilities.f90.

C.4.2.2 calc_det_3d()

```
integer recursive function num_utilities::calc_det::calc_det_3d (
                real(dp), dimension(:,:,:), intent(inout) deta,
                real(dp), dimension(:,:,:,:), intent(in) a,
                integer, intent(in) n)
```

private array version

Parameters

in,out	*deta*	output
in	*a*	input
in	*n*	size of matrix

Definition at line 542 of file num_utilities.f90.

Here is the call graph for this function:

C.5 eq_ops::calc_eq Interface Reference

Calculate the equilibrium quantities on a grid determined by straight field lines.

Public Member Functions

- integer function calc_eq_1 (grid_eq, eq)
  flux version
- integer function calc_eq_2 (grid_eq, eq_1, eq_2, dealloc_vars)
  metric version

C.5.1 Detailed Description

Calculate the equilibrium quantities on a grid determined by straight field lines.

This grid has the dimensions (n_par,loc_n_r).

Optionally, for eq_2, the used variables can be deallocated on the fly, to limit memory usage.

Returns

ierr

Definition at line 48 of file eq_ops.f90.

C.5.2 Member Function/Subroutine Documentation

C.5.2.1 calc_eq_1()

```
integer function eq_ops::calc_eq::calc_eq_1 (
            type(grid_type), intent(inout) grid_eq,
            type(eq_1_type), intent(inout) eq)
```

flux version

Parameters

in,out	*grid_eq*	equilibrium grid
in,out	*eq*	flux equilibrium variables

Definition at line 280 of file eq_ops.f90.

Here is the call graph for this function:

C.5.2.2 calc_eq_2()

```
integer function eq_ops::calc_eq::calc_eq_2 (
                type(grid_type), intent(inout) grid_eq,
                type(eq_1_type), intent(in) eq_1,
                type(eq_2_type), intent(inout) eq_2,
                logical, intent(in), optional dealloc_vars)
```

metric version

Parameters

in,out	*grid_eq*	equilibrium grid

in	*eq_1*	metric equilibrium variables
in,out	*eq_2*	metric equilibrium variables
in	*dealloc_vars*	deallocate variables on the fly after writing

Definition at line 434 of file eq_ops.f90.

Here is the call graph for this function:

C.6 grid_utilities::calc_eqd_grid Interface Reference

Calculate grid of equidistant points, where optionally the last point can be excluded.

Public Member Functions

- integer function calc_eqd_grid_1d (var, min_grid, max_grid, excl_last)
  1-D version
- integer function calc_eqd_grid_3d (var, min_grid, max_grid, grid_dim, excl_last)
  3-D version

C.6.1 Detailed Description

Calculate grid of equidistant points, where optionally the last point can be excluded.

Definition at line 75 of file grid_utilities.f90.

C.6.2 Member Function/Subroutine Documentation

C.6.2.1 calc_eqd_grid_1d()

```
integer function grid_utilities::calc_eqd_grid::calc_eqd_grid_1d (
                real(dp), dimension(:), intent(inout) var,
                real(dp), intent(in) min_grid,
                real(dp), intent(in) max_grid,
                logical, intent(in), optional excl_last)
```

1-D version

Parameters

in,out	*var*	output
in	*min_grid*	min. of angles [π]
in	*max_grid*	max. of angles [π]
in	*excl_last*	.true. if last point excluded

Definition at line 603 of file grid_utilities.f90.

C.6.2.2 calc_eqd_grid_3d()

```
integer function grid_utilities::calc_eqd_grid::calc_eqd_grid_3d (
                real(dp), dimension(:,:,:), intent(inout) var,
                real(dp), intent(in) min_grid,
                real(dp), intent(in) max_grid,
                integer, intent(in) grid_dim,
                logical, intent(in), optional excl_last)
```

3-D version

Parameters

in,out	*var*	output
in	*min_grid*	min. of angles [π]
in	*max_grid*	max. of angles [π]
in	*grid_dim*	in which dimension to create the grid
in	*excl_last*	.true. if last point excluded

Definition at line 527 of file grid_utilities.f90.

C.7 eq_utilities::calc_f_derivs Interface Reference

Transforms derivatives of the equilibrium quantities in E coordinates to derivatives in the F coordinates.

Public Member Functions

- integer function calc_f_derivs_1 (grid_eq, eq)
  flux version
- integer function calc_f_derivs_2 (eq)
  metric version

C.7.1 Detailed Description

Transforms derivatives of the equilibrium quantities in E coordinates to derivatives in the F coordinates.

Returns

ierr

Definition at line 36 of file eq_utilities.f90.

C.7.2 Member Function/Subroutine Documentation

C.7.2.1 calc_f_derivs_1()

```
integer function eq_utilities::calc_f_derivs::calc_f_derivs_1 (
                type(grid_type), intent(in) grid_eq,
                type(eq_1_type), intent(inout) eq)
```

flux version

Parameters

in	*grid_eq*	equilibrium grid
in,out	*eq*	flux equilibrium variables

Definition at line 754 of file eq_utilities.f90.

Here is the call graph for this function:

C.7.2.2 calc_f_derivs_2()

```
integer function eq_utilities::calc_f_derivs::calc_f_derivs_2 (
                type(eq_2_type), intent(inout) eq)
```

metric version

Parameters

in,out	*eq*	metric equilibrium variables

Definition at line 837 of file eq_utilities.f90.

Here is the call graph for this function:

C.8 eq_ops::calc_g_c Interface Reference

Calculate the lower metric elements in the C(ylindrical) coordinate system.

Public Member Functions

- integer function calc_g_c_ind (eq, deriv)
  individual version
- integer function calc_g_c_arr (eq, deriv)
  array version

C.8.1 Detailed Description

Calculate the lower metric elements in the C(ylindrical) coordinate system.

This is done directly using the formula's in [15]

Returns

ierr

Definition at line 139 of file eq_ops.f90.

C.8.2 Member Function/Subroutine Documentation

C.8.2.1 calc_g_c_arr()

```
integer function eq_ops::calc_g_c::calc_g_c_arr (
                type(eq_2_type), intent(inout) eq,
                integer, dimension(:,:), intent(in) deriv)
```

array version

Parameters

in,out	*eq*	metric equilibrium
in	*deriv*	derivatives

Definition at line 2857 of file eq_ops.f90.

C.8.2.2 calc_g_c_ind()

```
integer function eq_ops::calc_g_c::calc_g_c_ind (
                type(eq_2_type), intent(inout) eq,
                integer, dimension(:), intent(in) deriv)
```

individual version

Parameters

in,out	*eq*	metric equilibrium

in	*deriv*	derivatives

Definition at line 2826 of file eq_ops.f90.

Here is the call graph for this function:

C.9 eq_ops::calc_g_f Interface Reference

Calculate the metric coefficients in the F(lux) coordinate system.

Public Member Functions

- integer function calc_g_f_ind (eq, deriv)
  individual version
- integer function calc_g_f_arr (eq, deriv)
  array version

C.9.1 Detailed Description

Calculate the metric coefficients in the F(lux) coordinate system.

This is done using the metric coefficients in the equilibrium coordinate system and the transformation matrices.

Returns

ierr

Definition at line 183 of file eq_ops.f90.

C.9.2 Member Function/Subroutine Documentation

C.9.2.1 calc_g_f_arr()

```
integer function eq_ops::calc_g_f::calc_g_f_arr (
               type(eq_2_type), intent(inout) eq,
               integer, dimension(:,:), intent(in) deriv)
```

array version

Parameters

in,out	*eq*	metric equilibrium

in	*deriv*	derivatives

Definition at line 3113 of file eq_ops.f90.

C.9.2.2 calc_g_f_ind()

```
integer function eq_ops::calc_g_f::calc_g_f_ind (
                type(eq_2_type), intent(inout) eq,
                integer, dimension(:), intent(in) deriv)
```

individual version

Parameters

in,out	*eq*	metric equilibrium
in	*deriv*	derivatives

Definition at line 3090 of file eq_ops.f90.

Here is the call graph for this function:

C.10 eq_ops::calc_g_h Interface Reference

Calculate the lower metric coefficients in the equilibrium H(ELENA) coordinate system.

Public Member Functions

- integer function calc_g_h_ind (grid_eq, eq, deriv)

  individual version
- integer function calc_g_h_arr (grid_eq, eq, deriv)

  array version

C.10.1 Detailed Description

Calculate the lower metric coefficients in the equilibrium H(ELENA) coordinate system.

This is done using the HELENA output

Returns

ierr

Definition at line 169 of file eq_ops.f90.

C.10.2 Member Function/Subroutine Documentation

C.10.2.1 calc_g_h_arr()

```
integer function eq_ops::calc_g_h::calc_g_h_arr (
                type(grid_type), intent(in) grid_eq,
                type(eq_2_type), intent(inout) eq,
                integer, dimension(:,:), intent(in) deriv)
```

array version

Parameters

in	*grid_eq*	equilibirum grid
in,out	*eq*	metric equilibrium
in	*deriv*	derivatives

Definition at line 3069 of file eq_ops.f90.

C.10.2.2 calc_g_h_ind()

```
integer function eq_ops::calc_g_h::calc_g_h_ind (
                type(grid_type), intent(in) grid_eq,
                type(eq_2_type), intent(inout) eq,
                integer, dimension(:), intent(in) deriv)
```

individual version

Parameters

in	*grid_eq*	equilibrium grid
in,out	*eq*	metric equilibrium

in	*deriv*	derivatives

Definition at line 2919 of file eq_ops.f90.

Here is the call graph for this function:

C.11 eq_ops::calc_g_v Interface Reference

Calculate the lower metric coefficients in the equilibrium V(MEC) coordinate system.

Public Member Functions

- integer function calc_g_v_ind (eq, deriv)
  individual version
- integer function calc_g_v_arr (eq, deriv)
  array version

C.11.1 Detailed Description

Calculate the lower metric coefficients in the equilibrium V(MEC) coordinate system.

This is done using the metric coefficients in the C(ylindrical) coordinate system and the transformation matrices

Note

It is assumed that the lower order derivatives have been calculated already. If not, the results will be incorrect.

Returns

ierr

Definition at line 156 of file eq_ops.f90.

C.11.2 Member Function/Subroutine Documentation

C.11.2.1 calc_g_v_arr()

```
integer function eq_ops::calc_g_v::calc_g_v_arr (
                type(eq_2_type), intent(inout) eq,
                integer, dimension(:,:), intent(in) deriv)
```

array version

Parameters

in,out	*eq*	metric equilibrium
in	*deriv*	derivatives

Definition at line 2899 of file eq_ops.f90.

C.11.2.2 calc_g_v_ind()

```
integer function eq_ops::calc_g_v::calc_g_v_ind (
                type(eq_2_type), intent(inout) eq,
                integer, dimension(:), intent(in) deriv)
```

individual version

Parameters

in,out	*eq*	metric equilibrium
in	*deriv*	derivatives

Definition at line 2877 of file eq_ops.f90.

Here is the call graph for this function:

C.12 num_utilities::calc_int Interface Reference

Integrates a function using the trapezoidal rule.

Public Member Functions

- integer function calc_int_eqd (var, step_size, var_int)
  equidistant version
- integer function calc_int_reg (var, x, var_int)
  regular version

C.12.1 Detailed Description

Integrates a function using the trapezoidal rule.

This function can be defined on an equidistant grid or a regular one:

- For an equidistant grid,
$$\int_1^n f(x)dx = \sum_{k=1}^{n-1} (f(k+1) + f(k)) \frac{\Delta_x}{2},$$
is used, with n the number of points, which are assumed to be equidistant with a given step size $Delta_x$.
- For a regular grid,
$$\int_1^n f(x)dx = \sum_{k=1}^{n-1} (f(k+1) + f(k)) \frac{x(k+1) - x(k)}{2},$$
is used, with n the number of points.
  - Therefore, n points have to be specified as well as n values for the function to be integrated.
  - They have to be given in ascending order but the step size does not have to be constant.
  - This yields the following difference formula:
$$\int_1^n f(x)dx = \int_1^{n-1} f(x)dx + (f(n) + f(n-1)) \frac{x(n) - x(n-1)}{2},$$
which is used here

Note

For periodic function, the trapezoidal rule works well only if the last point of the grid is included, i.e. the point where the function is equal to the first point.

Returns

ierr

Definition at line 160 of file num_utilities.f90.

C.12.2 Member Function/Subroutine Documentation

C.12.2.1 calc_int_eqd()

```
integer function num_utilities::calc_int::calc_int_eqd (
            real(dp), dimension(:), intent(in) var,
            real(dp), intent(in) step_size,
            real(dp), dimension(:), intent(inout) var_int)
```

equidistant version

Parameters

in,out	*var_int*	integrated variable

in	*var*	variable to be integrated
in	*step_size*	step size of abscissa

Definition at line 326 of file num_utilities.f90.

Here is the call graph for this function:

C.12.2.2 calc_int_reg()

```
integer function num_utilities::calc_int::calc_int_reg (
                real(dp), dimension(:), intent(in) var,
                real(dp), dimension(:), intent(in) x,
                real(dp), dimension(:), intent(inout) var_int)
```

regular version

Parameters

in,out	*var_int*	integrated variable
in	*var*	variable to be integrated
in	*x*	abscissa

Definition at line 285 of file num_utilities.f90.

Here is the call graph for this function:

C.13 num_utilities::calc_inv Interface Reference

Calculate inverse of square matrix A.

Public Member Functions

- integer function calc_inv_0d (inv_0d, a)
  private constant version
- integer function calc_inv_3d (inv_3d, a, n)
  array version

C.13.1 Detailed Description

Calculate inverse of square matrix A.

This matrix can be defined on a 3-D grid or constant. The storage convention described in eq_vars.eq_2_type is used.

In the former case the size of the matrix (last two indices) should be small, as direct inversion is performed using Cramer's rule.

In the latter case, lapack routines are used.

Returns

ierr

Definition at line 81 of file num_utilities.f90.

C.13.2 Member Function/Subroutine Documentation

C.13.2.1 calc_inv_0d()

```
integer function num_utilities::calc_inv::calc_inv_0d (
                real(dp), dimension(:,:), intent(inout) inv_0d,
                real(dp), dimension(:,:), intent(in) a)
```

private constant version

Parameters

in,out	*inv_0d*	output
in	*a*	input

Definition at line 771 of file num_utilities.f90.

C.13.2.2 calc_inv_3d()

```
integer function num_utilities::calc_inv::calc_inv_3d (
                real(dp), dimension(:,:,:,:), intent(inout) inv_3d,
                real(dp), dimension(:,:,:,:), intent(in) a,
                integer, intent(in) n)
```

array version

Parameters

in,out	*inv_3d*	output

in	*a*	input
in	*n*	size of matrix

Definition at line 675 of file num_utilities.f90.

Here is the call graph for this function:

C.14 eq_utilities::calc_inv_met Interface Reference

Calculate $D_1^{m_1} D_2^{m_2} D_3^{m_3} X$ from $D_1^{i_1} D_2^{i_2} D_3^{i_3} X$ and $D_1^{j_1} D_2^{j_2} D_3^{j_3} Y$ where $XY = 1$ and $i, j = 0 \ldots m$, according to [15].

Public Member Functions

- integer function calc_inv_met_ind (x, y, deriv)

  individual matrix version
- integer function calc_inv_met_arr (x, y, deriv)
- integer function calc_inv_met_ind_0d (x, y, deriv)

  individual scalar version
- integer function calc_inv_met_arr_0d (x, y, deriv)

C.14.1 Detailed Description

Calculate $D_1^{m_1} D_2^{m_2} D_3^{m_3} X$ from $D_1^{i_1} D_2^{i_2} D_3^{i_3} X$ and $D_1^{j_1} D_2^{j_2} D_3^{j_3} Y$ where $XY = 1$ and $i, j = 0 \ldots m$, according to [15].

$D_1^{m_1} D_2^{m_2} D_3^{m_3}$ is defined as $\left(\frac{\partial}{\partial u^1}\right)^{m_1} \left(\frac{\partial}{\partial u^2}\right)^{m_2} \left(\frac{\partial}{\partial u^3}\right)^{m_3}$

Note

It is assumed that the lower order derivatives have been calculated already. If not, the results will be incorrect.

Returns

ierr

Definition at line 61 of file eq_utilities.f90.

C.14.2 Member Function/Subroutine Documentation

C.14.2.1 calc_inv_met_arr()

```
integer function eq_utilities::calc_inv_met::calc_inv_met_arr (
                real(dp), dimension(1:,1:,1:,1:,0:,0:,0:), intent(inout) x,
                real(dp), dimension(1:,1:,1:,1:,0:,0:,0:), intent(in) y,
                integer, dimension(:,:), intent(in) deriv)
```

Parameters

in	*deriv*	derivatives

Definition at line 321 of file eq_utilities.f90.

C.14.2.2 calc_inv_met_arr_0d()

```
integer function eq_utilities::calc_inv_met::calc_inv_met_arr_0d (
                real(dp), dimension(1:,1:,1:,0:,0:,0:), intent(inout) x,
                real(dp), dimension(1:,1:,1:,0:,0:,0:), intent(in) y,
                integer, dimension(:,:), intent(in) deriv)
```

Parameters

in	*deriv*	derivatives

Definition at line 341 of file eq_utilities.f90.

C.14.2.3 calc_inv_met_ind()

```
integer function eq_utilities::calc_inv_met::calc_inv_met_ind (
                real(dp), dimension(1:,1:,1:,1:,0:,0:,0:), intent(inout) x,
                real(dp), dimension(1:,1:,1:,1:,0:,0:,0:), intent(in) y,
                integer, dimension(:), intent(in) deriv)
```

individual matrix version

Parameters

in	*deriv*	derivatives

Definition at line 131 of file eq_utilities.f90.

Here is the call graph for this function:

C.14.2.4 calc_inv_met_ind_0d()

```
integer function eq_utilities::calc_inv_met::calc_inv_met_ind_0d (
                real(dp), dimension(1:,1:,1:,0:,0:,0:), intent(inout) x,
                real(dp), dimension(1:,1:,1:,0:,0:,0:), intent(in) y,
                integer, dimension(:), intent(in) deriv)
```

individual scalar version

Parameters

in	*deriv*	derivatives

Definition at line 260 of file eq_utilities.f90.

C.15 eq_ops::calc_jac_f Interface Reference

Calculate $\mathcal{J}_F$, the jacobian in Flux coordinates.

Public Member Functions

- integer function calc_jac_f_ind (eq, deriv)
  individual version
- integer function calc_jac_f_arr (eq, deriv)
  array version

C.15.1 Detailed Description

Calculate $\mathcal{J}_{\mathsf{F}}$, the jacobian in Flux coordinates.

This is done directly from

$$\mathcal{J}_{\mathsf{F}} = \det\left(\overline{\mathsf{T}}_{\mathsf{F}}^{\mathsf{E}}\right)\mathcal{J}_{\mathsf{E}}$$

Note

It is assumed that the lower order derivatives have been calculated already. If not, the results will be incorrect.

Returns

ierr

Definition at line 232 of file eq_ops.f90.

C.15.2 Member Function/Subroutine Documentation

C.15.2.1 calc_jac_f_arr()

```
integer function eq_ops::calc_jac_f::calc_jac_f_arr (
                type(eq_2_type), intent(inout) eq,
                integer, dimension(:,:), intent(in) deriv)
```

array version

Parameters

in,out	*eq*	metric equilibrium
in	*deriv*	derivatives

Definition at line 3330 of file eq_ops.f90.

C.15.2.2 calc_jac_f_ind()

```
integer function eq_ops::calc_jac_f::calc_jac_f_ind (
                type(eq_2_type), intent(inout) eq,
                integer, dimension(:), intent(in) deriv)
```

individual version

Parameters

in,out	*eq*	metric equilibrium

in	*deriv*	derivatives

Definition at line 3303 of file eq_ops.f90.

Here is the call graph for this function:

C.16 eq_ops::calc_jac_h Interface Reference

Calculate $\mathcal{J}_{\mathsf{H}}$, the jacobian in HELENA coordinates.

Public Member Functions

- integer function calc_jac_h_ind (grid_eq, eq_1, eq_2, deriv)
  individual version
- integer function calc_jac_h_arr (grid_eq, eq_1, eq_2, deriv)
  array version

C.16.1 Detailed Description

Calculate $\mathcal{J}_{\mathsf{H}}$, the jacobian in HELENA coordinates.

This is done directly from

$$\mathcal{J}_{\mathsf{H}} = q\frac{R^2}{F}$$

Note

It is assumed that the lower order derivatives have been calculated already. If not, the results will be incorrect.

Returns

ierr

Definition at line 213 of file eq_ops.f90.

C.16.2 Member Function/Subroutine Documentation

C.16.2.1 calc_jac_h_arr()

```
integer function eq_ops::calc_jac_h::calc_jac_h_arr (
                type(grid_type), intent(in) grid_eq,
                type(eq_1_type), intent(in) eq_1,
                type(eq_2_type), intent(inout) eq_2,
                integer, dimension(:,:), intent(in) deriv)
```

array version

Parameters

in	*grid_eq*	equilibrim grid
in	*eq_1*	flux equilibrium
in,out	*eq_2*	metric equilibrium
in	*deriv*	derivatives

Definition at line 3281 of file eq_ops.f90.

C.16.2.2 calc_jac_h_ind()

```
integer function eq_ops::calc_jac_h::calc_jac_h_ind (
                type(grid_type), intent(in) grid_eq,
                type(eq_1_type), intent(in) eq_1,
                type(eq_2_type), intent(inout) eq_2,
                integer, dimension(:), intent(in) deriv)
```

individual version

Parameters

in	*grid_eq*	equilibrium grid
in	*eq_1*	flux equilibrium
in,out	*eq_2*	metric equilibrium
in	*deriv*	derivatives

Definition at line 3185 of file eq_ops.f90.

Here is the call graph for this function:

C.17 eq_ops::calc_jac_v Interface Reference

Calculate $\mathcal{J}_V$, the jacobian in V(MEC) coordinates.

Public Member Functions

- integer function calc_jac_v_ind (grid, eq, deriv)
  individual version
- integer function calc_jac_v_arr (grid, eq, deriv)
  array version

C.17.1 Detailed Description

Calculate $\mathcal{J}_V$, the jacobian in V(MEC) coordinates.

This is done using VMEC output.

Returns

ierr

Definition at line 196 of file eq_ops.f90.

C.17.2 Member Function/Subroutine Documentation

C.17.2.1 calc_jac_v_arr()

```
integer function eq_ops::calc_jac_v::calc_jac_v_arr (
                type(grid_type), intent(in) grid,
                type(eq_2_type), intent(inout) eq,
                integer, dimension(:,:), intent(in) deriv)
```

array version

Parameters

in	*grid*	grid for which to calculate Jacobian

in,out	*eq*	metric equilibrium
in	*deriv*	derivatives

Definition at line 3164 of file eq_ops.f90.

C.17.2.2 calc_jac_v_ind()

```
integer function eq_ops::calc_jac_v::calc_jac_v_ind (
                type(grid_type), intent(in) grid,
                type(eq_2_type), intent(inout) eq,
                integer, dimension(:), intent(in) deriv)
```

individual version

Parameters

in	*grid*	grid for which to calculate Jacobian
in,out	*eq*	metric equilibrium
in	*deriv*	derivatives

Definition at line 3133 of file eq_ops.f90.

C.18 num_utilities::calc_mult Interface Reference

Calculate matrix multiplication of two square matrices $\overline{AB} = \overline{A}\,\overline{B}$.

Public Member Functions

- integer function calc_mult_0d_real (a, b, ab, n, transp)
  real constant version
- integer function calc_mult_3d_real (a, b, ab, n, transp)
  real array version
- integer function calc_mult_3d_complex (a, b, ab, n, transp)
  complex array version

C.18.1 Detailed Description

Calculate matrix multiplication of two square matrices $\overline{AB} = \overline{A}\,\overline{B}$.

This matrix can be defined on a 3-D grid or constant. The storage convention described in eq_vars.eq_2_type is used.

Returns

ierr

Definition at line 95 of file num_utilities.f90.

C.18.2 Member Function/Subroutine Documentation

C.18.2.1 calc_mult_0d_real()

```
integer function num_utilities::calc_mult::calc_mult_0d_real (
                real(dp), dimension(:), intent(in) a,
                real(dp), dimension(:), intent(in) b,
                real(dp), dimension(:), intent(inout) ab,
                integer, intent(in) n,
                logical, dimension(2), intent(in), optional transp)
```

real constant version

Parameters

in	*a*	input A
in	*b*	input B
in,out	*ab*	output A B
in	*n*	size of matrix
in	*transp*	whether A and/or B transposed

Definition at line 983 of file num_utilities.f90.

C.18.2.2 calc_mult_3d_complex()

```
integer function num_utilities::calc_mult::calc_mult_3d_complex (
                complex(dp), dimension(:,:,:,:), intent(in) a,
                complex(dp), dimension(:,:,:,:), intent(in) b,
                complex(dp), dimension(:,:,:,:), intent(inout) ab,
                integer, intent(in) n,
                logical, dimension(2), intent(in), optional transp)
```

complex array version

Parameters

in	*a*	input A
in	*b*	input B
in,out	*ab*	output A B
in	*n*	size of matrix
in	*transp*	whether A and/or B transposed

Definition at line 897 of file num_utilities.f90.

Here is the call graph for this function:

C.18.2.3 calc_mult_3d_real()

```
integer function num_utilities::calc_mult::calc_mult_3d_real (
                real(dp), dimension(:,:,:,:), intent(in) a,
                real(dp), dimension(:,:,:,:), intent(in) b,
                real(dp), dimension(:,:,:,:), intent(inout) ab,
                integer, intent(in) n,
                logical, dimension(2), intent(in), optional transp)
```

real array version

Parameters

in	*a*	input A
in	*b*	input B
in,out	*ab*	output A B
in	*n*	size of matrix
in	*transp*	whether A and/or B transposed

Definition at line 816 of file num_utilities.f90.

Here is the call graph for this function:

C.19 eq_ops::calc_rzl Interface Reference

Calculate R, Z & λ and derivatives in VMEC coordinates.

Public Member Functions

- integer function calc_rzl_ind (grid_eq, eq, deriv)
  individual version
- integer function calc_rzl_arr (grid_eq, eq, deriv)
  array version

C.19.1 Detailed Description

Calculate R, Z & λ and derivatives in VMEC coordinates.

This is done at the grid points given by the variables `theta_E` and `zeta_E` (contained in `trigon_factors`) and at every normal point.

The derivatives are indicated by the variable `deriv` which has 3 indices

Note

There is no HELENA equivalent because for HELENA simulations, R and Z are not necessary for calculation of the metric coefficients, and λ does not exist.

See also

calc_trigon_factors()

Returns

ierr

Definition at line 126 of file eq_ops.f90.

C.19.2 Member Function/Subroutine Documentation

C.19.2.1 calc_rzl_arr()

```
integer function eq_ops::calc_rzl::calc_rzl_arr (
                type(grid_type), intent(in) grid_eq,
                type(eq_2_type), intent(inout) eq,
                integer, dimension(:,:), intent(in) deriv)
```

array version

Parameters

in	*grid_eq*	equilibrium grid

in,out	*eq*	metric equilibrium
in	*deriv*	derivatives

Definition at line 2805 of file eq_ops.f90.

C.19.2.2 calc_rzl_ind()

```
integer function eq_ops::calc_rzl::calc_rzl_ind (
                type(grid_type), intent(in) grid_eq,
                type(eq_2_type), intent(inout) eq,
                integer, dimension(3), intent(in) deriv)
```

individual version

Parameters

in	*grid_eq*	equilibrium grid
in,out	*eq*	metric equilibrium
in	*deriv*	derivatives

Definition at line 2767 of file eq_ops.f90.

C.20 eq_ops::calc_t_hf Interface Reference

Calculate $\overline{\overline{T}}^{F}_{H}$, the transformation matrix between H(ELENA) and F(lux) coordinate systems.

Public Member Functions

- integer function calc_t_hf_ind (grid_eq, eq_1, eq_2, deriv)
  individual version
- integer function calc_t_hf_arr (grid_eq, eq_1, eq_2, deriv)
  array version

C.20.1 Detailed Description

Calculate $\overline{\overline{T}}^{F}_{H}$, the transformation matrix between H(ELENA) and F(lux) coordinate systems.

This is done directly using the formula's in [15]

Returns

ierr

Definition at line 271 of file eq_ops.f90.

C.20.2 Member Function/Subroutine Documentation

C.20.2.1 calc_t_hf_arr()

```
integer function eq_ops::calc_t_hf::calc_t_hf_arr (
                type(grid_type), intent(in) grid_eq,
                type(eq_1_type), intent(in) eq_1,
                type(eq_2_type), intent(inout) eq_2,
                integer, dimension(:,:), intent(in) deriv)
```

array version

Parameters

in	*grid_eq*	equilibrium grid
in	*eq_1*	flux equilibrium
in,out	*eq_2*	metric equilibrium
in	*deriv*	derivatives

Definition at line 3779 of file eq_ops.f90.

C.20.2.2 calc_t_hf_ind()

```
integer function eq_ops::calc_t_hf::calc_t_hf_ind (
                type(grid_type), intent(in) grid_eq,
                type(eq_1_type), intent(in) eq_1,
                type(eq_2_type), intent(inout) eq_2,
                integer, dimension(:), intent(in) deriv)
```

individual version

Parameters

in	*grid_eq*	equilibrium grid
in	*eq_1*	flux equilibrium
in,out	*eq_2*	metric equilibrium
in	*deriv*	derivatives

Definition at line 3606 of file eq_ops.f90.

Here is the call graph for this function:

C.21 eq_ops::calc_t_vc Interface Reference

Calculate $\overline{\overline{T}}_C^V$, the transformation matrix between C(ylindrical) and V(mec) coordinate system.

Public Member Functions

- integer function calc_t_vc_ind (eq, deriv)
  individual version
- integer function calc_t_vc_arr (eq, deriv)
  array version

C.21.1 Detailed Description

Calculate $\overline{\overline{T}}_C^V$, the transformation matrix between C(ylindrical) and V(mec) coordinate system.

This is done directly using the formula's in [15]

Returns

ierr

Definition at line 245 of file eq_ops.f90.

C.21.2 Member Function/Subroutine Documentation

C.21.2.1 calc_t_vc_arr()

```
integer function eq_ops::calc_t_vc::calc_t_vc_arr (
            type(eq_2_type), intent(inout) eq,
            integer, dimension(:,:), intent(in) deriv)
```

array version

Parameters

in,out	*eq*	metric equilibrium

in	*deriv*	derivatives

Definition at line 3393 of file eq_ops.f90.

C.21.2.2 calc_t_vc_ind()

```
integer function eq_ops::calc_t_vc::calc_t_vc_ind (
                type(eq_2_type), intent(inout) eq,
                integer, dimension(:), intent(in) deriv)
```

individual version

Parameters

in,out	*eq*	metric equilibrium
in	*deriv*	derivatives

Definition at line 3350 of file eq_ops.f90.

Here is the call graph for this function:

C.22 eq_ops::calc_t_vf Interface Reference

Calculate $\overline{\overline{T}}^{F}_{V}$, the transformation matrix between V(MEC) and F(lux) coordinate systems.

Public Member Functions

- integer function calc_t_vf_ind (grid_eq, eq_1, eq_2, deriv)
  individual version
- integer function calc_t_vf_arr (grid_eq, eq_1, eq_2, deriv)
  array version

C.22.1 Detailed Description

Calculate $\overline{\overline{T}}^{F}_{V}$, the transformation matrix between V(MEC) and F(lux) coordinate systems.

This is done directly using the formula's in [15]

Returns

ierr

Definition at line 258 of file eq_ops.f90.

C.22.2 Member Function/Subroutine Documentation

C.22.2.1 calc_t_vf_arr()

```
integer function eq_ops::calc_t_vf::calc_t_vf_arr (
                type(grid_type), intent(in) grid_eq,
                type(eq_1_type), intent(in) eq_1,
                type(eq_2_type), intent(inout) eq_2,
                integer, dimension(:,:), intent(in) deriv)
```

array version

Parameters

in	*grid_eq*	equilibrium grid
in	*eq_1*	flux equilibrium
in,out	*eq_2*	metric equilibrium
in	*deriv*	derivatives

Definition at line 3584 of file eq_ops.f90.

C.22.2.2 calc_t_vf_ind()

```
integer function eq_ops::calc_t_vf::calc_t_vf_ind (
                type(grid_type), intent(in) grid_eq,
                type(eq_1_type), intent(in) eq_1,
                type(eq_2_type), intent(inout) eq_2,
                integer, dimension(:), intent(in) deriv)
```

individual version

Parameters

in	*grid_eq*	equilibrium grid

in	*eq_1*	flux equilibrium
in,out	*eq_2*	metric equilibrium
in	*deriv*	derivatives

Definition at line 3413 of file eq_ops.f90.

Here is the call graph for this function:

C.23 grid_utilities::calc_tor_diff Interface Reference

Calculates the toroidal difference for a magnitude calculated on three toroidal points: two extremities and one in the middle.

Public Member Functions

- integer function calc_tor_diff_0d (v_mag, theta, norm_disc_prec, absolute, r)
  0-D version
- integer function calc_tor_diff_2d (v_com, theta, norm_disc_prec, absolute, r)
  2-D version

C.23.1 Detailed Description

Calculates the toroidal difference for a magnitude calculated on three toroidal points: two extremities and one in the middle.

This is done using the formula

$$\frac{b-a}{b+a},$$

for the relative difference between a and b. This is useful when it is used to calculate a toroidal ripple and these are the extreme points.

The procedure also needs the map between the flux poloidal angle and the geometrical poloidal angle.

In a first step the quantity is interpolated on an equidistant grid in the geometrical poloidal angle.

The difference is then calculated for values of constant geometric poloidal angle.

Finally, this result is transformed back to the Flux coordinates.

Note

1. The theta map should have first and last point equal.
2. This routine should be used only for periodic quantities (so not for some of the Flux quantities that are defined along generally non-rational magnetic field lines).

Returns

ierr

Definition at line 109 of file grid_utilities.f90.

C.23.2 Member Function/Subroutine Documentation

C.23.2.1 calc_tor_diff_0d()

```
integer function grid_utilities::calc_tor_diff::calc_tor_diff_0d (
                real(dp), dimension(:,:,:), intent(inout) v_mag,
                real(dp), dimension(:,:,:), intent(in) theta,
                integer, intent(in) norm_disc_prec,
                logical, intent(in), optional absolute,
                real(dp), dimension(:), intent(in), optional r)
```

0-D version

Parameters

in,out	*v_mag*	magnitude (dim1,dim2,dim3)
in	*theta*	geometric poloidal angle
in	*norm_disc_prec*	precision for normal derivatives
in	*absolute*	calculate absolute, not relative, difference
in	*r*	normal positions for theta

Definition at line 744 of file grid_utilities.f90.

C.23.2.2 calc_tor_diff_2d()

```
integer function grid_utilities::calc_tor_diff::calc_tor_diff_2d (
                real(dp), dimension(:,:,:,:,:), intent(inout) v_com,
                real(dp), dimension(:,:,:), intent(in) theta,
                integer, intent(in) norm_disc_prec,
                logical, intent(in), optional absolute,
                real(dp), dimension(:), intent(in), optional r)
```

2-D version

Parameters

in,out	*v_com*	covariant and contravariant components (dim1,dim2,dim3,3,2)

in	*theta*	geometric poloidal angle
in	*norm_disc_prec*	precision for normal derivatives
in	*absolute*	calculate absolute, not relative, difference
in	*r*	normal positions for theta

Definition at line 631 of file grid_utilities.f90.

Here is the call graph for this function:

C.24 x_ops::calc_x Interface Reference

Calculates either vectorial or tensorial perturbation variables.

Public Member Functions

- integer function calc_x_1 (mds, grid_eq, grid_x, eq_1, eq_2, x, lim_sec_x)
  vectorial version
- integer function calc_x_2 (mds, grid_eq, grid_x, eq_1, eq_2, x_a, x_b, x, lim_sec_x)
  tensorial version

C.24.1 Detailed Description

Calculates either vectorial or tensorial perturbation variables.

Optionally, the secondary mode number can be specified (`m` if poloidal flux is used as normal coordinate and `c n` if toroidal flux). By default, they are taken from the global `X_vars` variables.

Returns

ierr

Definition at line 39 of file X_ops.f90.

C.24.2 Member Function/Subroutine Documentation

C.24.2.1 calc_x_1()

```
integer function x_ops::calc_x::calc_x_1 (
            type(modes_type), intent(in) mds,
            type(grid_type), intent(in) grid_eq,
            type(grid_type), intent(in) grid_x,
            type(eq_1_type), intent(in) eq_1,
            type(eq_2_type), intent(in) eq_2,
            type(x_1_type), intent(inout) x,
            integer, dimension(2), intent(in), optional lim_sec_x)
```

vectorial version

Parameters

in	*mds*	general modes variables
in	*grid_eq*	equilibrium grid variables
in	*grid_x*	perturbation grid variables
in	*eq_1*	flux equilibrium
in	*eq_2*	metric equilibrium
in,out	*x*	vectorial perturbation variables
in	*lim_sec↩ _x*	limits of m_X (pol. flux) or \ n_X (tor. flux)

Definition at line 94 of file X_ops.f90.

Here is the call graph for this function:

C.24.2.2 calc_x_2()

```
integer function x_ops::calc_x::calc_x_2 (
            type(modes_type), intent(in) mds,
            type(grid_type), intent(in) grid_eq,
            type(grid_type), intent(in) grid_x,
            type(eq_1_type), intent(in) eq_1,
```

```
type(eq_2_type), intent(in) eq_2,
type(x_1_type), intent(inout) x_a,
type(x_1_type), intent(inout) x_b,
type(x_2_type), intent(inout) x,
integer, dimension(2,2), intent(in), optional lim_sec_x)
```

tensorial version

Parameters

in	*mds*	general modes variables
in	*grid_eq*	equilibrium grid variables
in	*grid_x*	perturbation grid variables
in	*eq_1*	flux equilibrium
in	*eq_2*	metric equilibrium
in,out	*x_a*	vectorial perturbation variables (dimension 1)
in,out	*x_b*	vectorial perturbation variables (dimension 2)
in,out	*x*	tensorial perturbation variables
in	*lim_sec↩ _x*	limits of m_X (pol flux) or n_X (tor flux) for both dimensions

Definition at line 129 of file X_ops.f90.

Here is the call graph for this function:

C.25 sol_utilities::calc_xuq Interface Reference

Calculates the normal ($\cdot_n$) or geodesic ($\cdot_g$) components of the plasma perturbation $\vec{\xi}$ or the magnetic perturbation $\vec{Q} = \nabla \times \left(\vec{x} \times \vec{B}\right)$.

Public Member Functions

- integer function calc_xuq_ind (mds_x, mds_sol, grid_eq, grid_x, grid_sol, eq_1, eq_2, x, sol, x_id, xuq_↩ style, time, xuq, deriv)

  (time) individual version
- integer function calc_xuq_arr (mds_x, mds_sol, grid_eq, grid_x, grid_sol, eq_1, eq_2, x, sol, x_id, xuq_↩ style, time, xuq, deriv)

  (time) array version

C.25.1 Detailed Description

Calculates the normal ($\cdot_n$) or geodesic ($\cdot_g$) components of the plasma perturbation $\vec{\xi}$ or the magnetic perturbation $\vec{Q} = \nabla \times \left(\vec{x} \times \vec{B}\right)$.

Which variables are plot depends on `XUQ_style` [15] :

- `XUQ_style` = 1: X (supports parallel derivative)
- `XUQ_style` = 2: U (supports parallel derivative)
- `XUQ_style` = 3: Q_n
- `XUQ_style` = 4: Q_g

X and U support the calculation of the parallel derivative.

For Q_n and Q_g, the metric variables have to be provided as well.

The perturbation grid is assumed to have the same angular coordinates as the equilibrium grid, and the normal coordinates correspond to either the equilibrium grid (`X_grid_style` 1), the solution grid (`X_grid_style` 2) or the enriched equilibrium grid (`X_grid_style` 3).

The output grid, furthermore, has the angular part corresponding to the equilibrium grid, and the normal part to the solution grid.

Returns

ierr

Definition at line 56 of file sol_utilities.f90.

C.25.2 Member Function/Subroutine Documentation

C.25.2.1 calc_xuq_arr()

```
integer function sol_utilities::calc_xuq::calc_xuq_arr (
                type(modes_type), intent(in), target mds_x,
                type(modes_type), intent(in), target mds_sol,
                type(grid_type), intent(in) grid_eq,
                type(grid_type), intent(in) grid_x,
                type(grid_type), intent(in) grid_sol,
                type(eq_1_type), intent(in) eq_1,
                type(eq_2_type), intent(in) eq_2,
                type(x_1_type), intent(in), target x,
                type(sol_type), intent(in) sol,
                integer, intent(in) x_id,
                integer, intent(in) xuq_style,
                real(dp), dimension(:), intent(in) time,
                complex(dp), dimension(:,:,:,:), intent(inout) xuq,
                logical, intent(in), optional deriv)
```

(time) array version

Parameters

in	*mds_x*	general modes variables in perturbation grid

in	*mds_sol*	general modes variables in solution grid
in	*grid_eq*	equilibrium grid
in	*grid_x*	perturbation grid
in	*grid_sol*	solution grid
in	*eq_1*	flux equilibrium
in	*eq_2*	metric equilibrium
in	*x*	perturbation variables
in	*sol*	solution variables
in	*x_id*	nr. of Eigenvalue
in	*xuq_style*	whether to calculate X, U, Q_n or Q_g
in	*time*	time range
in,out	*xuq*	X, U, Q_n or Q_g
in	*deriv*	return parallel derivative

Definition at line 65 of file sol_utilities.f90.

Here is the call graph for this function:

C.25.2.2 calc_xuq_ind()

```
integer function sol_utilities::calc_xuq::calc_xuq_ind (
                type(modes_type), intent(in) mds_x,
                type(modes_type), intent(in) mds_sol,
                type(grid_type), intent(in) grid_eq,
                type(grid_type), intent(in) grid_x,
```

```
type(grid_type), intent(in) grid_sol,
type(eq_1_type), intent(in) eq_1,
type(eq_2_type), intent(in) eq_2,
type(x_1_type), intent(in) x,
type(sol_type), intent(in) sol,
integer, intent(in) x_id,
integer, intent(in) xuq_style,
real(dp), intent(in) time,
complex(dp), dimension(:,:,:), intent(inout) xuq,
logical, intent(in), optional deriv)
```

(time) individual version

Parameters

in	*mds_x*	general modes variables in perturbation grid
in	*mds_sol*	general modes variables in solution grid
in	*grid_eq*	equilibrium grid
in	*grid_x*	perturbation grid
in	*grid_sol*	solution grid
in	*eq_1*	flux equilibrium
in	*eq_2*	metric equilibrium
in	*x*	perturbation variables
in	*sol*	solution variables
in	*x_id*	nr. of Eigenvalue
in	*xuq_style*	whether to calculate X, U, Qn or Qg
in,out	*xuq*	normal component of perturbation
in	*deriv*	return parallel derivative

Definition at line 518 of file sol_utilities.f90.

C.26 num_ops::calc_zero_hh Interface Reference

Finds the zero of a function using Householder iteration.

Public Member Functions

- character(len=max_str_ln) function calc_zero_hh_0d (zero, fun, ord, guess, max_nr_backtracks, output)

  0-D version
- character(len=max_str_ln) function calc_zero_hh_3d (dims, zero, fun, ord, guess, max_nr_backtracks, output)

  3-D version

C.26.1 Detailed Description

Finds the zero of a function using Householder iteration.

If something goes wrong, by default multiple tries can be attempted, by backtracking on the correction by multiplying it by a relaxation factor. This can be done `max_nr_backtracks` times.

If still nothing is achieved, an error message is returned, that is empty otherwise.

Definition at line 35 of file num_ops.f90.

C.26.2 Member Function/Subroutine Documentation

C.26.2.1 calc_zero_hh_0d()

```
character(len=max_str_ln) function num_ops::calc_zero_hh::calc_zero_hh_0d (
                real(dp), intent(inout) zero,
                external real(dp) function(real(dp), intent(in) x, integer, intent(in) ord) fun,
                integer, intent(in) ord,
                real(dp), intent(in) guess,
                integer, intent(in), optional max_nr_backtracks,
                logical, intent(in), optional output)
```

0-D version

Parameters

`in,out`	*fun*	`fun(x,ord)` with • `x` abscissa • `ord` order of derivative • `fun` ordinate
`in,out`	*zero*	output
`in`	*ord*	order of solution
`in`	*guess*	first guess
`in`	*max_nr_backtracks*	max nr. of tries with different relaxation factors
`in`	*output*	give output on convergence

Returns

possible error message

Definition at line 48 of file num_ops.f90.

Here is the call graph for this function:

C.26.2.2 calc_zero_hh_3d()

```
character(len=max_str_ln) function num_ops::calc_zero_hh::calc_zero_hh_3d (
                integer, dimension(3), intent(in) dims,
                real(dp), dimension(dims(1),dims(2),dims(3)), intent(inout) zero,
                external real(dp) function, dimension(dims(1),dims(2),dims(3))(integer, dimension(3), intent(in)
dims, real(dp), dimension(dims(1),dims(2),dims(3)), intent(in) x, integer, intent(in) ord) fun,
                integer, intent(in) ord,
                real(dp), dimension(dims(1),dims(2),dims(3)), intent(in) guess,
                integer, intent(in), optional max_nr_backtracks,
                logical, intent(in), optional output)
```

3-D version

Parameters

in,out	*fun*	fun(dims,x,ord) with • dims(3) dimension of abscissa and ordinate • x(dims(1),dims(2),dims(3)) abscissa • ord order of derivative • fun(dims(1),dims(2),dims(3)) ordinate
in	*dims*	dimensions of the problem
in,out	*zero*	output
in	*ord*	order of solution
in	*guess*	first guess

in	*max_nr_↩ backtracks*	max nr. of backtracks
in	*output*	give output on convergence

Returns

possible error message

Definition at line 241 of file num_ops.f90.

Here is the call graph for this function:

C.27 num_utilities::con Interface Reference

Either takes the complex conjugate of a square matrix element A, defined on a 3-D grid, or not.

Public Member Functions

- complex(dp) function, dimension(d(1), d(2), d(3)) con_3d (a, c, sym, d)

  3-D version
- complex(dp) function, dimension(d(1), d(2)) con_2d (a, c, sym, d)

  2-D version
- complex(dp) function, dimension(d(1)) con_1d (a, c, sym, d)

  1-D version
- complex(dp) function con_0d (a, c, sym)

  0-D version

C.27.1 Detailed Description

Either takes the complex conjugate of a square matrix element A, defined on a 3-D grid, or not.

This is done depending on whether the indices of the matrix element correspond to the upper (conjugate) or lower (no conjugate) triangular part.

Note

Only for symmetric matrices does this have to be applied.

Definition at line 221 of file num_utilities.f90.

C.27.2 Member Function/Subroutine Documentation

C.27.2.1 con_0d()

```
complex(dp) function num_utilities::con::con_0d (
                complex(dp), intent(in) a,
                integer, dimension(2), intent(in) c,
                logical, intent(in) sym)
```

0-D version

Parameters

in	*a*	input A
in	*c*	indices in square matrix
in	*sym*	if A is symmetric

Returns

output B

Definition at line 1489 of file num_utilities.f90.

Here is the call graph for this function:

C.27.2.2 con_1d()

```
complex(dp) function, dimension(d(1)) num_utilities::con::con_1d (
                complex(dp), dimension(:), intent(in) a,
                integer, dimension(2), intent(in) c,
                logical, intent(in) sym,
                integer, dimension(1), intent(in) d)
```

1-D version

Parameters

in	*a*	input A

in	*c*	indices in square matrix
in	*sym*	if A is symmetric
in	*d*	dimensions of matrix A

Returns

output B

Definition at line 1474 of file num_utilities.f90.

Here is the call graph for this function:

C.27.2.3 con_2d()

```
complex(dp) function, dimension(d(1),d(2)) num_utilities::con::con_2d (
                complex(dp), dimension(:,:), intent(in) a,
                integer, dimension(2), intent(in) c,
                logical, intent(in) sym,
                integer, dimension(2), intent(in) d)
```

2-D version

Parameters

in	*a*	input A
in	*c*	indices in square matrix
in	*sym*	if A is symmetric
in	*d*	dimensions of matrix A

Returns

output B

Definition at line 1459 of file num_utilities.f90.

Here is the call graph for this function:

C.27.2.4 con_3d()

```
complex(dp) function, dimension(d(1),d(2),d(3)) num_utilities::con::con_3d (
                complex(dp), dimension(:,:,:), intent(in) a,
                integer, dimension(2), intent(in) c,
                logical, intent(in) sym,
                integer, dimension(3), intent(in) d)
```

3-D version

Parameters

in	*a*	input A
in	*c*	indices in square matrix
in	*sym*	if A is symmetric
in	*d*	dimensions of matrix A

Returns

output B

Definition at line 1444 of file num_utilities.f90.

Here is the call graph for this function:

C.28 num_utilities::con2dis Interface Reference

Convert between points from a continuous grid to a discrete grid.

Public Member Functions

- integer function con2dis_eqd (pt_c, pt_d, lim_c, lim_d)
  equidistant version
- integer function con2dis_reg (pt_c, pt_d, var_c)
  regular version

C.28.1 Detailed Description

Convert between points from a continuous grid to a discrete grid.

This is done by providing either the the limits on the grid (`lim_c` and `lim_d`), in which case the grid is assumed to be equidistant, or the grid values themselves, in which case the grid just has to be regular.

The output is a real value where the floored integer is the index in the discrete grid and the remainder corresponds to the fraction towards the next index. If no solution is found, a negative value is outputted, as well as a message.

Returns

ierr

Definition at line 188 of file num_utilities.f90.

C.28.2 Member Function/Subroutine Documentation

C.28.2.1 con2dis_eqd()

```
integer function num_utilities::con2dis::con2dis_eqd (
            real(dp), intent(in) pt_c,
            real(dp), intent(inout) pt_d,
            real(dp), dimension(2), intent(in) lim_c,
            integer, dimension(2), intent(in) lim_d)
```

equidistant version

Parameters

in	*pt_c*	point on continous grid

in,out	*pt_d*	point on discrete grid
in	*lim↩ _c*	[min_c,max_c]
in	*lim↩ _d*	[min_d,max_d]

Definition at line 1205 of file num_utilities.f90.

C.28.2.2 con2dis_reg()

```
integer function num_utilities::con2dis::con2dis_reg (
                real(dp), intent(in) pt_c,
                real(dp), intent(inout) pt_d,
                real(dp), dimension(:), intent(in) var_c)
```

regular version

Parameters

in	*pt_c*	point on continous grid
in,out	*pt_d*	point on discrete grid
in	*var↩ _c*	continous grid values

Definition at line 1225 of file num_utilities.f90.

C.29 pb3d_utilities::conv_1d2nd Interface Reference

Converts 1-D to n-D variables.

Public Member Functions

- subroutine conv_1d2nd_1d (var_in, var_out)
  1-D version
- subroutine conv_1d2nd_2d (var_in, var_out)
  2-D version
- subroutine conv_1d2nd_3d (var_in, var_out)
  3-D version
- subroutine conv_1d2nd_4d (var_in, var_out)
  4-D version
- subroutine conv_1d2nd_6d (var_in, var_out)
  6-D version
- subroutine conv_1d2nd_7d (var_in, var_out)
  7-D version

C.29.1 Detailed Description

Converts 1-D to n-D variables.

The 1-D variables are used for internal storage in HDF5, whereas the n_D variables correspond to the ones in PB3D.

Definition at line 21 of file PB3D_utilities.f90.

C.29.2 Member Function/Subroutine Documentation

C.29.2.1 conv_1d2nd_1d()

```
subroutine pb3d_utilities::conv_1d2nd::conv_1d2nd_1d (
            type(var_1d_type), intent(in) var_in,
            real(dp), dimension(:), intent(inout), allocatable var_out)
```

1-D version

Parameters

in	*var_in*	1D variable
in,out	*var_out*	output variable

Definition at line 38 of file PB3D_utilities.f90.

C.29.2.2 conv_1d2nd_2d()

```
subroutine pb3d_utilities::conv_1d2nd::conv_1d2nd_2d (
            type(var_1d_type), intent(in) var_in,
            real(dp), dimension(:,:), intent(inout), allocatable var_out)
```

2-D version

Parameters

in	*var_in*	1D variable
in,out	*var_out*	output variable

Definition at line 49 of file PB3D_utilities.f90.

C.29.2.3 conv_1d2nd_3d()

```
subroutine pb3d_utilities::conv_1d2nd::conv_1d2nd_3d (
            type(var_1d_type), intent(in) var_in,
            real(dp), dimension(:,:,:), intent(inout), allocatable var_out)
```

3-D version

Parameters

in	*var_in*	1D variable

in,out	*var_out*	output variable

Definition at line 64 of file PB3D_utilities.f90.

C.29.2.4 conv_1d2nd_4d()

```
subroutine pb3d_utilities::conv_1d2nd::conv_1d2nd_4d (
            type(var_1d_type), intent(in) var_in,
            real(dp), dimension(:,:,:,:), intent(inout), allocatable var_out)
```

4-D version

Parameters

in	*var_in*	1D variable
in,out	*var_out*	output variable

Definition at line 78 of file PB3D_utilities.f90.

C.29.2.5 conv_1d2nd_6d()

```
subroutine pb3d_utilities::conv_1d2nd::conv_1d2nd_6d (
            type(var_1d_type), intent(in) var_in,
            real(dp), dimension(:,:,:,:,:,:), intent(inout), allocatable var_out)
```

6-D version

Parameters

in	*var_in*	1D variable
in,out	*var_out*	output variable

Definition at line 93 of file PB3D_utilities.f90.

C.29.2.6 conv_1d2nd_7d()

```
subroutine pb3d_utilities::conv_1d2nd::conv_1d2nd_7d (
            type(var_1d_type), intent(in) var_in,
            real(dp), dimension(:,:,:,:,:,:,:), intent(inout), allocatable var_out)
```

7-D version

Parameters

in	*var_in*	1D variable

in,out	*var_out*	output variable

Definition at line 110 of file PB3D_utilities.f90.

C.30 num_utilities::conv_mat Interface Reference

Converts a (symmetric) matrix A with the storage convention described in eq_vars.eq_2_type.

Public Member Functions

- integer function conv_mat_3d (a, b, n, transp)
  version defined on 3-D grid
- integer function conv_mat_3d_complex (a, b, n, transp)
  complex version defined on 3D grid
- integer function conv_mat_0d (a, b, n, transp)
  scalar version
- integer function conv_mat_0d_complex (a, b, n, transp)
  complex scalar version

C.30.1 Detailed Description

Converts a (symmetric) matrix A with the storage convention described in eq_vars.eq_2_type.

This matric can have elements depending on a 3-D grid or be constant.

- If the matrix is stored with n^2 numbers, only the lower diagonal elements are kept in matrix B.
- If only the lower diagonal elements are stored, they are copied to the upper diagonal ones of the matrix B as well.

Optionally, the transpose can be calculated.

Note

1. The routine does not check whether the matrix is indeed symmetric and that information may thus be lost after conversion.
2. This routine makes a copy of A so that by providing A as input argument for both A and B overwrites A.

Returns

ierr

Definition at line 123 of file num_utilities.f90.

C.30.2 Member Function/Subroutine Documentation

C.30.2.1 conv_mat_0d()

```
integer function num_utilities::conv_mat::conv_mat_0d (
                real(dp), dimension(:), intent(in) a,
                real(dp), dimension(:), intent(inout) b,
                integer, intent(in) n,
                logical, intent(in), optional transp)
```

scalar version

Parameters

in	*a*	matrix to be converted

in,out	*b*	converted matrix
in	*n*	size of matrix
in	*transp*	transpose

Definition at line 1083 of file num_utilities.f90.

C.30.2.2 conv_mat_0d_complex()

```
integer function num_utilities::conv_mat::conv_mat_0d_complex (
                complex(dp), dimension(:), intent(in) a,
                complex(dp), dimension(:), intent(inout) b,
                integer, intent(in) n,
                logical, intent(in), optional transp)
```

complex scalar version

Parameters

in	*a*	matrix to be converted
in,out	*b*	converted matrix
in	*n*	size of matrix
in	*transp*	transpose

Definition at line 1178 of file num_utilities.f90.

C.30.2.3 conv_mat_3d()

```
integer function num_utilities::conv_mat::conv_mat_3d (
                real(dp), dimension(:,:,:,:), intent(in) a,
                real(dp), dimension(:,:,:,:), intent(inout) b,
                integer, intent(in) n,
                logical, intent(in), optional transp)
```

version defined on 3-D grid

Parameters

in	*a*	matrix to be converted
in,out	*b*	converted matrix
in	*n*	size of matrix
in	*transp*	transpose

Definition at line 1014 of file num_utilities.f90.

Here is the call graph for this function:

C.30.2.4 conv_mat_3d_complex()

```
integer function num_utilities::conv_mat::conv_mat_3d_complex (
                complex(dp), dimension(:,:,:,:), intent(in) a,
                complex(dp), dimension(:,:,:,:), intent(inout) b,
                integer, intent(in) n,
                logical, intent(in), optional transp)
```

complex version defined on 3D grid

Parameters

in	*a*	matrix to be converted
in,out	*b*	converted matrix
in	*n*	size of matrix
in	*transp*	transpose

Definition at line 1109 of file num_utilities.f90.

Here is the call graph for this function:

C.31 grid_utilities::coord_e2f Interface Reference

Converts Equilibrium coordinates $(r, \theta, \zeta)_{\mathsf{E}}$. to Flux coordinates $(r, \theta, \zeta)_{\mathsf{F}}$.

Public Member Functions

- integer function coord_e2f_r (grid_eq, r_e, r_f, r_e_array, r_f_array)
  version with only r
- integer function coord_e2f_rtz (grid_eq, r_e, theta_e, zeta_e, r_f, theta_f, zeta_f, r_e_array, r_f_array)
  version with r, theta and zeta

C.31.1 Detailed Description

Converts Equilibrium coordinates $(r, \theta, \zeta)_{\mathsf{E}}$. to Flux coordinates $(r, \theta, \zeta)_{\mathsf{F}}$.

Optionally, two arrays `r_E_array` and `r_F_array` can be provided, that define the mapping between the both coordinate system.

Standard, for E, the poloidal or toroidal normalized flux is used and for F, the poloidal or toroidal flux in F coordinates, divided by 2π.

Returns

ierr

Definition at line 66 of file grid_utilities.f90.

C.31.2 Member Function/Subroutine Documentation

C.31.2.1 coord_e2f_r()

```
integer function grid_utilities::coord_e2f::coord_e2f_r (
                type(grid_type), intent(in) grid_eq,
                real(dp), dimension(:), intent(in) r_e,
                real(dp), dimension(:), intent(inout) r_f,
                real(dp), dimension(:), intent(in), optional, target r_e_array,
                real(dp), dimension(:), intent(in), optional, target r_f_array)
```

version with only r

Parameters

in	*grid_eq*	equilibrium grid (for normal local limits)

in	*r_e*	(local) r_E
in,out	*r_f*	(local) r_F
in	*r_e_array*	optional array that defines mapping between two coord. systems
in	*r_f_array*	optional array that defines mapping between two coord. systems

Definition at line 471 of file grid_utilities.f90.

C.31.2.2 coord_e2f_rtz()

```
integer function grid_utilities::coord_e2f::coord_e2f_rtz (
                type(grid_type), intent(in) grid_eq,
                real(dp), dimension(:), intent(in) r_e,
                real(dp), dimension(:,:,:), intent(in) theta_e,
                real(dp), dimension(:,:,:), intent(in) zeta_e,
                real(dp), dimension(:), intent(inout) r_f,
                real(dp), dimension(:,:,:), intent(inout) theta_f,
                real(dp), dimension(:,:,:), intent(inout) zeta_f,
                real(dp), dimension(:), intent(in), optional, target r_e_array,
                real(dp), dimension(:), intent(in), optional, target r_f_array)
```

version with r, theta and zeta

Parameters

in	*grid_eq*	equilibrium grid (for normal local limits)
in	*r_e*	(local) r_E
in	*theta_e*	θ_E
in	*zeta_e*	ζ_E
in,out	*r_f*	(local) r_F
in,out	*theta_f*	θ_F
in,out	*zeta_f*	ζ_F
in	*r_e_array*	optional array that defines mapping between two coord. systems
in	*r_f_array*	optional array that defines mapping between two coord. systems

Definition at line 357 of file grid_utilities.f90.

C.32 grid_utilities::coord_f2e Interface Reference

Converts Flux coordinates $(r, \theta, \zeta)_{\mathrm{F}}$ to Equilibrium coordinates $(r, \theta, \zeta)_{\mathrm{E}}$.

Public Member Functions

- integer function coord_f2e_r (grid_eq, r_f, r_e, r_f_array, r_e_array)

  version with only r
- integer function coord_f2e_rtz (grid_eq, r_f, theta_f, zeta_f, r_e, theta_e, zeta_e, r_f_array, r_e_array, ord)

  version withi r, theta and zeta

C.32.1 Detailed Description

Converts Flux coordinates $(r, \theta, \zeta)_{\mathrm{F}}$ to Equilibrium coordinates $(r, \theta, \zeta)_{\mathrm{E}}$.

Optionally, two arrays `r_F_array` and `r_E_array` can be provided, that define the mapping between the both coordinate system.

Standard, for E, the poloidal or toroidal normalized flux is used and for F, the poloidal or toroidal flux in F coordinates, divided by 2π.

Note

For VMEC, it can be slow, as the zero of a non-linear expression must be sought. This is done currently using num_ops.calc_zero_hh().

Returns

ierr

Definition at line 47 of file grid_utilities.f90.

C.32.2 Member Function/Subroutine Documentation

C.32.2.1 coord_f2e_r()

```
integer function grid_utilities::coord_f2e::coord_f2e_r (
                type(grid_type), intent(in) grid_eq,
                real(dp), dimension(:), intent(in) r_f,
                real(dp), dimension(:), intent(inout) r_e,
                real(dp), dimension(:), intent(in), optional, target r_f_array,
                real(dp), dimension(:), intent(in), optional, target r_e_array)
```

version with only r

Parameters

in	*grid_eq*	equilibrium grid (for possibly loc_r)

in	*r_f*	(local) r_F
in,out	*r_e*	(local) r_E
in	*r_f_array*	optional array that defines mapping between two coord. systems
in	*r_e_array*	optional array that defines mapping between two coord. systems

Definition at line 301 of file grid_utilities.f90.

C.32.2.2 coord_f2e_rtz()

```
integer function grid_utilities::coord_f2e::coord_f2e_rtz (
                type(grid_type), intent(in) grid_eq,
                real(dp), dimension(:), intent(in) r_f,
                real(dp), dimension(:,:,:), intent(in) theta_f,
                real(dp), dimension(:,:,:), intent(in) zeta_f,
                real(dp), dimension(:), intent(inout) r_e,
                real(dp), dimension(:,:,:), intent(inout) theta_e,
                real(dp), dimension(:,:,:), intent(inout) zeta_e,
                real(dp), dimension(:), intent(in), optional, target r_f_array,
                real(dp), dimension(:), intent(in), optional, target r_e_array,
                integer, intent(in), optional ord)
```

version withi r, theta and zeta

Parameters

in	*grid_eq*	equilibrium grid (for normal local limits and possibly loc_r)
in	*r_f*	(local) r_F
in	*theta_f*	θ_F
in	*zeta_f*	ζ_F
in,out	*r_e*	(local) r_E
in,out	*theta_e*	θ_E
in,out	*zeta_e*	ζ_E
in	*r_f_array*	optional array that defines mapping between two coord. systems
in	*r_e_array*	optional array that defines mapping between two coord. systems
in	*ord*	order (only used for eq_style 1)

Definition at line 118 of file grid_utilities.f90.

Here is the call graph for this function:

C.33 hdf5_vars::dealloc_var_1d Interface Reference

Deallocates 1D variable.

Public Member Functions

- subroutine dealloc_var_1d_ind (var_1d)
  individual version
- subroutine dealloc_var_1d_arr (var_1d)
  array version
- subroutine dealloc_var_1d_arr_2 (var_1d)
  rank 2 array version

C.33.1 Detailed Description

Deallocates 1D variable.

Definition at line 68 of file HDF5_vars.f90.

C.33.2 Member Function/Subroutine Documentation

C.33.2.1 dealloc_var_1d_arr()

```
subroutine hdf5_vars::dealloc_var_1d::dealloc_var_1d_arr (
            type(var_1d_type), dimension(:), intent(inout), allocatable var_1d)
```

array version

Parameters

in,out	*var_1d*	array of 1D variables to be deallocated

Definition at line 153 of file HDF5_vars.f90.

C.33.2.2 dealloc_var_1d_arr_2()

```
subroutine hdf5_vars::dealloc_var_1d::dealloc_var_1d_arr_2 (
                type(var_1d_type), dimension(:,:), intent(inout), allocatable var_1d)
```

rank 2 array version

Parameters

in,out	*var_1d*	array of 1D variables to be deallocated

Definition at line 135 of file HDF5_vars.f90.

C.33.2.3 dealloc_var_1d_ind()

```
subroutine hdf5_vars::dealloc_var_1d::dealloc_var_1d_ind (
                type(var_1d_type), intent(out) var_1d)
```

individual version

Parameters

out	*var_1d*	1D variable to be deallocated

Definition at line 169 of file HDF5_vars.f90.

C.34 hdf5_vars::dealloc_xml_str Interface Reference

Deallocates XML_str_type.

Public Member Functions

- subroutine dealloc_xml_str_ind (xml_str)
  individual version
- subroutine dealloc_xml_str_arr (xml_str)
  array version

C.34.1 Detailed Description

Deallocates XML_str_type.

Definition at line 60 of file HDF5_vars.f90.

C.34.2 Member Function/Subroutine Documentation

C.34.2.1 dealloc_xml_str_arr()

```
subroutine hdf5_vars::dealloc_xml_str::dealloc_xml_str_arr (
            type(xml_str_type), dimension(:), intent(inout), allocatable xml_str)
```

array version

Definition at line 113 of file HDF5_vars.f90.

C.34.2.2 dealloc_xml_str_ind()

```
subroutine hdf5_vars::dealloc_xml_str::dealloc_xml_str_ind (
            type(xml_str_type), intent(out) xml_str)
```

individual version

Parameters

out	*xml_str*	XML string to be deallocated

Definition at line 129 of file HDF5_vars.f90.

C.35 num_utilities::dis2con Interface Reference

Convert between points from a discrete grid to a continuous grid.

Public Member Functions

- integer function dis2con_eqd (pt_d, pt_c, lim_d, lim_c)
  equidistant version
- integer function dis2con_reg (pt_d, pt_c, var_c)
  regular version

C.35.1 Detailed Description

Convert between points from a discrete grid to a continuous grid.

This is done by providing either the the limits on the grid (`lim_c` and `lim_d`), in which case the grid is assumed to be equidistant, or the grid values themselves, in which case the grid just has to be regular.

The output is a real value. If the discrete value lies outside the range, in the case of a regular grid, a negative value is outputted, as well as a message.

Returns

ierr

Definition at line 206 of file num_utilities.f90.

C.35.2 Member Function/Subroutine Documentation

C.35.2.1 dis2con_eqd()

```
integer function num_utilities::dis2con::dis2con_eqd (
            integer, intent(in) pt_d,
            real(dp), intent(inout) pt_c,
            integer, dimension(2), intent(in) lim_d,
            real(dp), dimension(2), intent(in) lim_c)
```

equidistant version

Parameters

in	*pt_d*	point on discrete grid
in,out	*pt_c*	point on continous grid
in	*lim↩ _d*	[min_d,max_d]
in	*lim↩ _c*	[min_c,max_c]

Definition at line 1335 of file num_utilities.f90.

C.35.2.2 dis2con_reg()

```
integer function num_utilities::dis2con::dis2con_reg (
            integer, intent(in) pt_d,
            real(dp), intent(inout) pt_c,
            real(dp), dimension(:), intent(in) var_c)
```

regular version

Parameters

in	*pt_d*	point on discrete grid
in,out	*pt_c*	point on continous grid
in	*var↩ _c*	continous grid values

Definition at line 1355 of file num_utilities.f90.

C.36 eq_vars::eq_1_type Type Reference

flux equilibrium type

Public Member Functions

- procedure init (eq, grid, setup_e, setup_f)
  initialize
- procedure copy (eq_i, grid_i, eq_o)
  copy
- procedure dealloc (eq)
  deallocate

Public Attributes

- real(dp), dimension(:,:), allocatable pres_e
  pressure, and norm. deriv.
- real(dp), dimension(:,:), allocatable q_saf_e
  safety factor
- real(dp), dimension(:,:), allocatable rot_t_e
  rot. transform
- real(dp), dimension(:,:), allocatable flux_p_e
  poloidal flux and norm. deriv.
- real(dp), dimension(:,:), allocatable flux_t_e
  toroidal flux and norm. deriv.
- real(dp), dimension(:,:), allocatable pres_fd
  pressure, and norm. deriv.
- real(dp), dimension(:,:), allocatable q_saf_fd
  safety factor
- real(dp), dimension(:,:), allocatable rot_t_fd
  rot. transform
- real(dp), dimension(:,:), allocatable flux_p_fd
  poloidal flux and norm. deriv.
- real(dp), dimension(:,:), allocatable flux_t_fd
  toroidal flux and norm. deriv.
- real(dp), dimension(:), allocatable rho
  density
- real(dp), dimension(2) estim_mem_usage
  expected memory usage

C.36.1 Detailed Description

flux equilibrium type

The arrays here are of the form

- `(r)` for variables without derivs.
- `(r,Dr)` for variables with derivs.

Definition at line 63 of file eq_vars.f90.

C.36.2 Member Function/Subroutine Documentation

C.36.2.1 copy()

```
procedure eq_vars::eq_1_type::copy (
            class(eq_1_type), intent(in) eq_i,
            type(grid_type), intent(in) grid_i,
            type(eq_1_type), intent(inout) eq_o)
```

copy

Parameters

in	*eq_i*	eq_1 to be copied
in	*grid↩ _i*	grid of eq_i
in,out	*eq_o*	copied eq_1

Definition at line 82 of file eq_vars.f90.

C.36.2.2 dealloc()

```
procedure eq_vars::eq_1_type::dealloc (
            class(eq_1_type), intent(inout) eq)
```

deallocate

Parameters

in,out	*eq*	equilibrium to be deallocated

Definition at line 84 of file eq_vars.f90.

C.36.2.3 init()

```
procedure eq_vars::eq_1_type::init (
            class(eq_1_type), intent(inout) eq,
            type(grid_type), intent(in) grid,
            logical, intent(in), optional setup_e,
            logical, intent(in), optional setup_f)
```

initialize

The normal and angular grid can be in any coord. system, as only the grid sizes are used, not the coordinate values.

Optionally, it can be chosen individually whether the E or F(D) quantities are allocated. D means that the derivatives are in F as well. The rationale behind this is that the E quantities are only used in the pre-perturbation phase.

Note

1. The E quantities are not written in eq_ops.print_output_eq().
2. The quantities that do not have a derivative are considered F quantities. Alternatively, all quantities that have only one version, are considered F quantities, such as `rho`, `kappa_n`, ...
3. The maximum derivative degree for flux quantities is one higher than `max_deriv`, because the derivative of some of them appear in the transformation matrices.

Parameters

`in,out`	*eq*	equilibrium to be initialized
`in`	*grid*	equilibrium grid
`in`	*setup↩ _e*	whether to set up E
`in`	*setup↩ _f*	whether to set up F

Definition at line 80 of file eq_vars.f90.

C.36.3 Member Data Documentation

C.36.3.1 estim_mem_usage

```
real(dp), dimension(2) eq_vars::eq_1_type::estim_mem_usage
```

expected memory usage

Note

Debug version only

Definition at line 76 of file eq_vars.f90.

C.36.3.2 flux_p_e

```
real(dp), dimension(:,:), allocatable eq_vars::eq_1_type::flux_p_e
```

poloidal flux and norm. deriv.

Definition at line 67 of file eq_vars.f90.

C.36.3.3 flux_p_fd

```
real(dp), dimension(:,:), allocatable eq_vars::eq_1_type::flux_p_fd
```

poloidal flux and norm. deriv.

Definition at line 72 of file eq_vars.f90.

C.36.3.4 flux_t_e

```
real(dp), dimension(:,:), allocatable eq_vars::eq_1_type::flux_t_e
```

toroidal flux and norm. deriv.

Definition at line 68 of file eq_vars.f90.

C.36.3.5 flux_t_fd

```
real(dp), dimension(:,:), allocatable eq_vars::eq_1_type::flux_t_fd
```

toroidal flux and norm. deriv.

Definition at line 73 of file eq_vars.f90.

C.36.3.6 pres_e

```
real(dp), dimension(:,:), allocatable eq_vars::eq_1_type::pres_e
```

pressure, and norm. deriv.

Definition at line 64 of file eq_vars.f90.

C.36.3.7 pres_fd

```
real(dp), dimension(:,:), allocatable eq_vars::eq_1_type::pres_fd
```

pressure, and norm. deriv.

Definition at line 69 of file eq_vars.f90.

C.36.3.8 q_saf_e

```
real(dp), dimension(:,:), allocatable eq_vars::eq_1_type::q_saf_e
```

safety factor

Definition at line 65 of file eq_vars.f90.

C.36.3.9 q_saf_fd

```
real(dp), dimension(:,:), allocatable eq_vars::eq_1_type::q_saf_fd
```

safety factor

Definition at line 70 of file eq_vars.f90.

C.36.3.10 rho

```
real(dp), dimension(:), allocatable eq_vars::eq_1_type::rho
```

density

Definition at line 74 of file eq_vars.f90.

C.36.3.11 rot_t_e

```
real(dp), dimension(:,:), allocatable eq_vars::eq_1_type::rot_t_e
```

rot. transform

Definition at line 66 of file eq_vars.f90.

C.36.3.12 rot_t_fd

```
real(dp), dimension(:,:), allocatable eq_vars::eq_1_type::rot_t_fd
```

rot. transform

Definition at line 71 of file eq_vars.f90.

C.37 eq_vars::eq_2_type Type Reference

metric equilibrium type

Public Member Functions

- procedure init (eq, grid, setup_e, setup_f)
  initialize
- procedure copy (eq_i, grid_i, eq_o)
  copy
- procedure dealloc (eq)
  deallocate

Public Attributes

- real(dp), dimension(:,:,:,:,:,:), allocatable r_e
  R in E(quilibrium) coord.
- real(dp), dimension(:,:,:,:,:,:), allocatable z_e
  Z in E(quilibrium) coords.
- real(dp), dimension(:,:,:,:,:,:), allocatable l_e
  L(ambda) in E(quilibrium) coords.
- real(dp), dimension(:,:,:,:,:,:,:), allocatable g_c
  in the C(ylindrical) coord. system
- real(dp), dimension(:,:,:,:,:,:,:), allocatable g_e
  in the E(quilibrium) coord. system
- real(dp), dimension(:,:,:,:,:,:,:), allocatable h_e
  in the E(quilibrium) coord. system
- real(dp), dimension(:,:,:,:,:,:,:), allocatable g_f
  in the F(lux) coord. system with derivs. in V(MEC) system
- real(dp), dimension(:,:,:,:,:,:,:), allocatable h_f
  in the F(lux) coord. system with derivs. in V(MEC) system
- real(dp), dimension(:,:,:,:,:,:,:), allocatable g_fd
  in the F(lux) coord. system with derivs in F(lux) system
- real(dp), dimension(:,:,:,:,:,:,:), allocatable h_fd
  in the F(lux) coord. system with derivs in F(lux) system
- real(dp), dimension(:,:,:,:,:,:,:), allocatable t_vc
  C(ylindrical) to V(MEC).
- real(dp), dimension(:,:,:,:,:,:,:), allocatable t_ef
  E(quilibrium) to F(lux).
- real(dp), dimension(:,:,:,:,:,:,:), allocatable t_fe
  F(lux) to E(quilibrium).
- real(dp), dimension(:,:,:,:,:,:), allocatable det_t_ef
  determinant of T_EF
- real(dp), dimension(:,:,:,:,:,:), allocatable det_t_fe
  determinant of T_FE
- real(dp), dimension(:,:,:,:,:,:), allocatable jac_e
  jacobian of E(quilibrium) coord. system
- real(dp), dimension(:,:,:,:,:,:), allocatable jac_f
  jacobian of F(lux) coord. system with derivs. in V(MEC) system
- real(dp), dimension(:,:,:,:,:,:), allocatable jac_fd
  jacobian of F(lux) coord. system with derivs. in F(lux) system
- real(dp), dimension(:,:,:), allocatable s
  magnetic shear
- real(dp), dimension(:,:,:), allocatable kappa_n
  normal curvature
- real(dp), dimension(:,:,:), allocatable kappa_g
  geodesic curvature
- real(dp), dimension(:,:,:), allocatable sigma
  parallel current
- real(dp), dimension(2) estim_mem_usage
  expected memory usage

C.37.1 Detailed Description

metric equilibrium type

The arrays here are of the form

- `(angle_1,angle_2,r)` for scalars without derivs.
- `(angle_1,angle_2,r,D123)` for scalars with derivs.
- `(angle_1,angle_2,r,6/9,D1,D2,D3)` for tensors with derivs.
  where it is refered to the discussion of the grid type for an explanation of the angles `angle_1` and `angle_2`.

The last index refers to the derivatives in coordinate 1, 2 and 3, which refer to the coordinates described in [15].

- For E(quilibrium) coordinates, they are $(r, \theta, \zeta)_{\mathsf{E}}$.
- For F(lux) coordinates, they are $(\alpha, \psi, \gamma)_{\mathsf{F}}$, where
  - $\alpha = \zeta - q\theta$ and $\gamma = \theta$ for pol. flux,
  - $\alpha = -\theta + \iota\zeta$ and $\gamma = \zeta$ Nfor tor. flux.

The fourth index for tensorial variables correspond to the 9 or 6 (symmetric) different values:

$$\begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix} \text{ or } \begin{pmatrix} 1 & & \\ 2 & 4 & \\ 3 & 5 & 6 \end{pmatrix}$$

Note

Fortran only allows for 7 dimensions in arrays.

Definition at line 114 of file eq_vars.f90.

C.37.2 Member Function/Subroutine Documentation

C.37.2.1 copy()

```
procedure eq_vars::eq_2_type::copy (
              class(eq_2_type), intent(in) eq_i,
              type(grid_type), intent(in) grid_i,
              type(eq_2_type), intent(inout) eq_o)
```

copy

Returns

ierr

Parameters

in	*eq_i*	eq_2 to be copied

in	*grid↩ _i*	grid of eq_i
in,out	*eq_o*	copied eq_1

Definition at line 150 of file eq_vars.f90.

C.37.2.2 dealloc()

```
procedure eq_vars::eq_2_type::dealloc (
            class(eq_2_type), intent(inout) eq)
```

deallocate

Parameters

in,out	*eq*	equilibrium to be deallocated

Definition at line 152 of file eq_vars.f90.

C.37.2.3 init()

```
procedure eq_vars::eq_2_type::init (
            class(eq_2_type), intent(inout) eq,
            type(grid_type), intent(in) grid,
            logical, intent(in), optional setup_e,
            logical, intent(in), optional setup_f)
```

initialize

See also

For explanation see init_eq_1().

Note

The maximum derivative degree for R, Z and lambda is one higher than `max_deriv`, because their first derivative already appears in g_C and h_C.

Parameters

in,out	*eq*	equilibrium to be initialized
in	*grid*	equilibrium grid
in	*setup↩ _e*	whether to set up E

in	*setup↩ _f*	whether to set up F

Definition at line 148 of file eq_vars.f90.

C.37.3 Member Data Documentation

C.37.3.1 det_t_ef

```
real(dp), dimension(:,:,:,:,:,:), allocatable eq_vars::eq_2_type::det_t_ef
```

determinant of T_EF

Definition at line 132 of file eq_vars.f90.

C.37.3.2 det_t_fe

```
real(dp), dimension(:,:,:,:,:,:), allocatable eq_vars::eq_2_type::det_t_fe
```

determinant of T_FE

Definition at line 133 of file eq_vars.f90.

C.37.3.3 estim_mem_usage

```
real(dp), dimension(2) eq_vars::eq_2_type::estim_mem_usage
```

expected memory usage

Note

Debug version only

Definition at line 144 of file eq_vars.f90.

C.37.3.4 g_c

```
real(dp), dimension(:,:,:,:,:,:,:), allocatable eq_vars::eq_2_type::g_c
```

in the C(ylindrical) coord. system

Definition at line 120 of file eq_vars.f90.

C.37.3.5 g_e

```
real(dp), dimension(:,:,:,:,:,:), allocatable eq_vars::eq_2_type::g_e
```

in the E(quilibrium) coord. system

Definition at line 121 of file eq_vars.f90.

C.37.3.6 g_f

```
real(dp), dimension(:,:,:,:,:,:), allocatable eq_vars::eq_2_type::g_f
```

in the F(lux) coord. system with derivs. in V(MEC) system

Definition at line 123 of file eq_vars.f90.

C.37.3.7 g_fd

```
real(dp), dimension(:,:,:,:,:,:), allocatable eq_vars::eq_2_type::g_fd
```

in the F(lux) coord. system with derivs in F(lux) system

Definition at line 125 of file eq_vars.f90.

C.37.3.8 h_e

```
real(dp), dimension(:,:,:,:,:,:), allocatable eq_vars::eq_2_type::h_e
```

in the E(quilibrium) coord. system

Definition at line 122 of file eq_vars.f90.

C.37.3.9 h_f

```
real(dp), dimension(:,:,:,:,:,:), allocatable eq_vars::eq_2_type::h_f
```

in the F(lux) coord. system with derivs. in V(MEC) system

Definition at line 124 of file eq_vars.f90.

C.37.3.10 h_fd

```
real(dp), dimension(:,:,:,:,:,:), allocatable eq_vars::eq_2_type::h_fd
```

in the F(lux) coord. system with derivs in F(lux) system

Definition at line 126 of file eq_vars.f90.

C.37.3.11 jac_e

```
real(dp), dimension(:,:,:,:,:), allocatable eq_vars::eq_2_type::jac_e
```

jacobian of E(quilibrium) coord. system

Definition at line 135 of file eq_vars.f90.

C.37.3.12 jac_f

```
real(dp), dimension(:,:,:,:,:), allocatable eq_vars::eq_2_type::jac_f
```

jacobian of F(lux) coord. system with derivs. in V(MEC) system

Definition at line 136 of file eq_vars.f90.

C.37.3.13 jac_fd

```
real(dp), dimension(:,:,:,:,:), allocatable eq_vars::eq_2_type::jac_fd
```

jacobian of F(lux) coord. system with derivs. in F(lux) system

Definition at line 137 of file eq_vars.f90.

C.37.3.14 kappa_g

```
real(dp), dimension(:,:,:), allocatable eq_vars::eq_2_type::kappa_g
```

geodesic curvature

Definition at line 141 of file eq_vars.f90.

C.37.3.15 kappa_n

```
real(dp), dimension(:,:,:), allocatable eq_vars::eq_2_type::kappa_n
```

normal curvature

Definition at line 140 of file eq_vars.f90.

C.37.3.16 l_e

```
real(dp), dimension(:,:,:,:,:), allocatable eq_vars::eq_2_type::l_e
```

L(ambda) in E(quilibrium) coords.

Definition at line 118 of file eq_vars.f90.

C.37.3.17 r_e

```
real(dp), dimension(:,:,:,:,:), allocatable eq_vars::eq_2_type::r_e
```

R in E(quilibrium) coord.

Definition at line 116 of file eq_vars.f90.

C.37.3.18 s

```
real(dp), dimension(:,:,:), allocatable eq_vars::eq_2_type::s
```

magnetic shear

Definition at line 139 of file eq_vars.f90.

C.37.3.19 sigma

```
real(dp), dimension(:,:,:), allocatable eq_vars::eq_2_type::sigma
```

parallel current

Definition at line 142 of file eq_vars.f90.

C.37.3.20 t_ef

```
real(dp), dimension(:,:,:,:,:,:), allocatable eq_vars::eq_2_type::t_ef
```

E(quilibrium) to F(lux).

Definition at line 129 of file eq_vars.f90.

C.37.3.21 t_fe

```
real(dp), dimension(:,:,:,:,:,:), allocatable eq_vars::eq_2_type::t_fe
```

F(lux) to E(quilibrium).

Definition at line 130 of file eq_vars.f90.

C.37.3.22 t_vc

```
real(dp), dimension(:,:,:,:,:,:), allocatable eq_vars::eq_2_type::t_vc
```

C(ylindrical) to V(MEC).

Definition at line 128 of file eq_vars.f90.

C.37.3.23 z_e

```
real(dp), dimension(:,:,:,:,:,:), allocatable eq_vars::eq_2_type::z_e
```

Z in E(quilibrium) coords.

Definition at line 117 of file eq_vars.f90.

C.38 vmec_utilities::fourier2real Interface Reference

Inverse Fourier transformation, from VMEC.

Public Member Functions

- integer function fourier2real_1 (varf_c, varf_s, trigon_factors, varr, sym, deriv)
  version with trigonometric factors
- integer function fourier2real_2 (varf_c, varf_s, theta, zeta, varr, sym, deriv)
  version with angles

C.38.1 Detailed Description

Inverse Fourier transformation, from VMEC.

Also calculates the poloidal or toroidal derivatives in VMEC coords., as indicated by the variable `deriv(2)`.

(The normal derivative is done on the variables in Fourier space, and should be provided here in `varf_i` if needed).

There are two variants:

1. version using trigon_factors, which is useful when the grid on which the trigonometric factors are defined is not regular and ideally when they are reused multiple times.
2. version using θ and ζ directly, which is useful for small, unique calculations.

Both these versions make use of a factor that represents angular derivatives. For `deriv = [j,k]`, this is:

- $m^j(-n)^k(-1)^{\frac{j+k+1}{2}}$ for `varf_c`,
- $m^j(-n)^k(-1)^{\frac{j+k}{2}}$ for `varf_s`,

where the divisions have to be done using integers, i.e. without remainder. The first two factors are straightforward, and the third one originates in the change of sign when deriving a cosine, but not for a sine.

Finally, depending on whether $(j+k)$ is even or odd, the correct cos or sin is chosen.

Returns

ierr

Definition at line 56 of file VMEC_utilities.f90.

C.38.2 Member Function/Subroutine Documentation

C.38.2.1 fourier2real_1()

```
integer function vmec_utilities::fourier2real::fourier2real_1 (
                real(dp), dimension(:,:), intent(in) varf_c,
                real(dp), dimension(:,:), intent(in) varf_s,
                real(dp), dimension(:,:,:,:,:), intent(in) trigon_factors,
                real(dp), dimension(:,:,:), intent(inout) varr,
                logical, dimension(2), intent(in), optional sym,
                integer, dimension(2), intent(in), optional deriv)
```

version with trigonometric factors

Parameters

in	*varf_c*	cos factor of variable in Fourier space
in	*varf_s*	sin factor of variable in Fourier space
in	*trigon_factors*	trigonometric factor cosine and sine at these angles (see calc_trigon_factors())
in,out	*varr*	variable in real space
in	*sym*	whether to use varf_c (1) and / or varf_s (2)
in	*deriv*	optional derivatives in angular coordinates

Definition at line 65 of file VMEC_utilities.f90.

C.38.2.2 fourier2real_2()

```
integer function vmec_utilities::fourier2real::fourier2real_2 (
                real(dp), dimension(:,:), intent(in) varf_c,
                real(dp), dimension(:,:), intent(in) varf_s,
                real(dp), dimension(:,:,:), intent(in) theta,
                real(dp), dimension(:,:,:), intent(in) zeta,
                real(dp), dimension(:,:,:), intent(inout) varr,
                logical, dimension(2), intent(in), optional sym,
                integer, dimension(2), intent(in), optional deriv)
```

version with angles

Parameters

in	*varf_c*	cos factor of variable in Fourier space
in	*varf_s*	sin factor of variable in Fourier space
in	*theta*	θ_E

in	*zeta*	ζ_E
in,out	*varr*	variable in real space
in	*sym*	whether to use varf_c (1) and / or varf_s (2)
in	*deriv*	optional derivatives in angular coordinates

Definition at line 163 of file VMEC_utilities.f90.

C.39 mpi_utilities::get_ghost_arr Interface Reference

Fill the ghost regions in an array.

Public Member Functions

- integer function get_ghost_arr_3d_complex (arr, size_ghost)
  3-D complex version
- integer function get_ghost_arr_3d_real (arr, size_ghost)
  3-D real version
- integer function get_ghost_arr_2d_complex (arr, size_ghost)
  2-D complex version
- integer function get_ghost_arr_1d_real (arr, size_ghost)
  1-D real version

C.39.1 Detailed Description

Fill the ghost regions in an array.

This is done by sending the first normal point of a process to the left process.

Every MPI message is identified by its sending process. The array should have the extended size, including ghost regions.

Returns

ierr

Definition at line 73 of file MPI_utilities.f90.

C.39.2 Member Function/Subroutine Documentation

C.39.2.1 get_ghost_arr_1d_real()

```
integer function mpi_utilities::get_ghost_arr::get_ghost_arr_1d_real (
                real(dp), dimension(:), intent(in) arr,
                integer, intent(in) size_ghost)
```

1-D real version

Parameters

in	*arr*	divided array

in	*size_↩ ghost*	width of ghost region

Definition at line 554 of file MPI_utilities.f90.

C.39.2.2 get_ghost_arr_2d_complex()

```
integer function mpi_utilities::get_ghost_arr::get_ghost_arr_2d_complex (
                complex(dp), dimension(:,:), intent(inout) arr,
                integer, intent(in) size_ghost)
```

2-D complex version

Parameters

in,out	*arr*	divided array
in	*size_↩ ghost*	width of ghost region

Definition at line 510 of file MPI_utilities.f90.

C.39.2.3 get_ghost_arr_3d_complex()

```
integer function mpi_utilities::get_ghost_arr::get_ghost_arr_3d_complex (
                complex(dp), dimension(:,:,:), intent(inout) arr,
                integer, intent(in) size_ghost)
```

3-D complex version

Parameters

in,out	*arr*	divided array
in	*size_↩ ghost*	width of ghost region

Definition at line 421 of file MPI_utilities.f90.

C.39.2.4 get_ghost_arr_3d_real()

```
integer function mpi_utilities::get_ghost_arr::get_ghost_arr_3d_real (
                real(dp), dimension(:,:,:), intent(inout) arr,
                integer, intent(in) size_ghost)
```

3-D real version

Parameters

in,out	*arr*	divided array

in	*size_↩ ghost*	width of ghost region

Definition at line 465 of file MPI_utilities.f90.

C.40 mpi_utilities::get_ser_var Interface Reference

Gather parallel variable in serial version on group master.

Public Member Functions

- integer function get_ser_var_complex (var, ser_var, scatter)
  complex version
- integer function get_ser_var_real (var, ser_var, scatter)
  real version
- integer function get_ser_var_int (var, ser_var, scatter)
  integer version

C.40.1 Detailed Description

Gather parallel variable in serial version on group master.

Optionally, all the processes receive the parallel variable using `scatter`.

Note

The serial variable has to be allocatable and if unallocated, it will be allocated.

Returns

ierr

Definition at line 55 of file MPI_utilities.f90.

C.40.2 Member Function/Subroutine Documentation

C.40.2.1 get_ser_var_complex()

```
integer function mpi_utilities::get_ser_var::get_ser_var_complex (
                complex(dp), dimension(:), intent(in) var,
                complex(dp), dimension(:), intent(inout), allocatable ser_var,
                logical, intent(in), optional scatter)
```

complex version

Parameters

in	*var*	parallel vector

in,out	*ser_var*	serial vector
in	*scatter*	optionally scatter the result to all the processes

Definition at line 106 of file MPI_utilities.f90.

C.40.2.2 get_ser_var_int()

```
integer function mpi_utilities::get_ser_var::get_ser_var_int (
                integer, dimension(:), intent(in) var,
                integer, dimension(:), intent(inout), allocatable ser_var,
                logical, intent(in), optional scatter)
```

integer version

Parameters

in	*var*	parallel vector
in,out	*ser_var*	serial vector
in	*scatter*	optionally scatter the result to all the processes

Definition at line 254 of file MPI_utilities.f90.

C.40.2.3 get_ser_var_real()

```
integer function mpi_utilities::get_ser_var::get_ser_var_real (
                real(dp), dimension(:), intent(in) var,
                real(dp), dimension(:), intent(inout), allocatable ser_var,
                logical, intent(in), optional scatter)
```

real version

Parameters

in	*var*	parallel vector
in,out	*ser_var*	serial vector
in	*scatter*	optionally scatter the result to all the processes

Definition at line 180 of file MPI_utilities.f90.

C.41 grid_vars::grid_type Type Reference

Type for grids.

Public Member Functions

- procedure init (grid, n, i_lim, divided)

  initialize
- procedure copy (grid_i, grid_o)

  copy
- procedure dealloc (grid)

  deallocate

Public Attributes

- integer, dimension(3) n

  tot nr. of points
- integer loc_n_r

  local nr. of normal points
- integer i_min

  min. normal index of this process in full arrays
- integer i_max

  max. normal index of this process in full arrays
- logical divided

  whether the grid is split up among the processes
- real(dp), dimension(:), pointer r_e => null()

  E(quilibrium) coord. values at n points.
- real(dp), dimension(:), pointer r_f => null()

  F(lux) coord. values at n points.
- real(dp), dimension(:), pointer loc_r_e => null()

  E(quilibrium) coord. values at n points.
- real(dp), dimension(:), pointer loc_r_f => null()

  F(lux) coord. values at n points.
- real(dp), dimension(:,:,:), pointer theta_e => null()

  E(quilibrium) coord. values of first angle at n points in 3-D.
- real(dp), dimension(:,:,:), pointer theta_f => null()

  F(lux) coord. values of first angle at n points in 3-D.
- real(dp), dimension(:,:,:), pointer zeta_e => null()

  E(quilibrium) coord. values of second angle at n points in 3-D.
- real(dp), dimension(:,:,:), pointer zeta_f => null()

  F(lux) coord. values of second angle at n points in 3-D.
- real(dp), dimension(:,:,:,:,:), allocatable trigon_factors

  trigonometric factor cosine for the inverse fourier transf.
- real(dp) estim_mem_usage

  estimated memory usage

C.41.1 Detailed Description

Type for grids.

The grids are saved in the following format: `(angle_1,angle_2,r)`, where `angle_1` and `angle_2` can be any angle that completely describe a flux surface.

For example, they can refer to a grid of θ and ζ values, but they can also refer to (Modified) flux coordinates with a parallel angle and a field line coordinate (α).

For field-aligned grids, `angle_1` is generally chosen equal to the parallel coordinate in the Flux coordinate system, and `angle_2` equal to the field line label (so the second dimension of the matrices is then chosen to be of size 1 for the calculations on a single field line).

At the same time, the parallel coordinate, in which integrations will have to be done, ocupies the first index. This is good for numerical efficiency.

For specific equilibrium grids, such as the case for HELENA equilibria, `angle_1` corresponds to θ and `angle_2` to the symmetry angle ζ.

It is important that this order of the coordinates in space is consistent among all the variables.

Definition at line 59 of file grid_vars.f90.

C.41.2 Member Function/Subroutine Documentation

C.41.2.1 copy()

```
procedure grid_vars::grid_type::copy (
            class(grid_type), intent(in) grid_i,
            type(grid_type), intent(inout) grid_o)
```

copy

Note

Does not copy possible trigoniometric factors.

Returns

ierr

Parameters

in	*grid_i*	grid to be copied
in,out	*grid_o*	copied grid

Definition at line 81 of file grid_vars.f90.

C.41.2.2 dealloc()

```
procedure grid_vars::grid_type::dealloc (
              class(grid_type), intent(inout) grid)
```

deallocate

Parameters

in,out	*grid*	grid to be deallocated

Definition at line 83 of file grid_vars.f90.

C.41.2.3 init()

```
procedure grid_vars::grid_type::init (
              class(grid_type), intent(inout) grid,
              integer, dimension(3), intent(in) n,
              integer, dimension(2), intent(in), optional i_lim,
              logical, intent(in), optional divided)
```

initialize

Optionally, the local limits can be provided for a divided grid.

Optionally, it can be set whether the grid is divided or not. A situation where this is useful is when only a subset of MPI processes is used to calculate the solution in SLEPC. In this case, the extra ranks all only contain the last grid point, and the last used process has as upper limit this same grid point. This way, all the procedures are reusable, but in this case if only one process is used, this procedure becomes confused and sets this process to undivided. In most cases, this functionality probably does not need to be used.

Returns

ierr

Parameters

in,out	*grid*	grid to be initialized
in	*n*	tot. nr. of points (par,r,alpha)
in	*i_lim*	min. and max. local normal index
in	*divided*	divided grid or not

Definition at line 79 of file grid_vars.f90.

C.41.3 Member Data Documentation

C.41.3.1 divided

```
logical grid_vars::grid_type::divided
```

whether the grid is split up among the processes

Definition at line 64 of file grid_vars.f90.

C.41.3.2 estim_mem_usage

```
real(dp) grid_vars::grid_type::estim_mem_usage
```

estimated memory usage

Note

Debug version only

Definition at line 75 of file grid_vars.f90.

C.41.3.3 i_max

```
integer grid_vars::grid_type::i_max
```

max. normal index of this process in full arrays

Definition at line 63 of file grid_vars.f90.

C.41.3.4 i_min

```
integer grid_vars::grid_type::i_min
```

min. normal index of this process in full arrays

Definition at line 62 of file grid_vars.f90.

C.41.3.5 loc_n_r

```
integer grid_vars::grid_type::loc_n_r
```

local nr. of normal points

Definition at line 61 of file grid_vars.f90.

C.41.3.6 loc_r_e

```
real(dp), dimension(:), pointer grid_vars::grid_type::loc_r_e => null()
```

E(quilibrium) coord. values at n points.

Definition at line 67 of file grid_vars.f90.

C.41.3.7 loc_r_f

```
real(dp), dimension(:), pointer grid_vars::grid_type::loc_r_f => null()
```

F(lux) coord. values at n points.

Definition at line 68 of file grid_vars.f90.

C.41.3.8 n

```
integer, dimension(3) grid_vars::grid_type::n
```

tot nr. of points

Definition at line 60 of file grid_vars.f90.

C.41.3.9 r_e

```
real(dp), dimension(:), pointer grid_vars::grid_type::r_e => null()
```

E(quilibrium) coord. values at n points.

Definition at line 65 of file grid_vars.f90.

C.41.3.10 r_f

```
real(dp), dimension(:), pointer grid_vars::grid_type::r_f => null()
```

F(lux) coord. values at n points.

Definition at line 66 of file grid_vars.f90.

C.41.3.11 theta_e

```
real(dp), dimension(:,:,:), pointer grid_vars::grid_type::theta_e => null()
```

E(quilibrium) coord. values of first angle at n points in 3-D.

Definition at line 69 of file grid_vars.f90.

C.41.3.12 theta_f

```
real(dp), dimension(:,:,:), pointer grid_vars::grid_type::theta_f => null()
```

F(lux) coord. values of first angle at n points in 3-D.

Definition at line 70 of file grid_vars.f90.

C.41.3.13 trigon_factors

```
real(dp), dimension(:,:,:,:,:), allocatable grid_vars::grid_type::trigon_factors
```

trigonometric factor cosine for the inverse fourier transf.

Definition at line 73 of file grid_vars.f90.

C.41.3.14 zeta_e

```
real(dp), dimension(:,:,:), pointer grid_vars::grid_type::zeta_e => null()
```

E(quilibrium) coord. values of second angle at n points in 3-D.

Definition at line 71 of file grid_vars.f90.

C.41.3.15 zeta_f

```
real(dp), dimension(:,:,:), pointer grid_vars::grid_type::zeta_f => null()
```

F(lux) coord. values of second angle at n points in 3-D.

Definition at line 72 of file grid_vars.f90.

C.42 hdf5_vars::hdf5_file_type Type Reference

HDF5 data type, containing the information about HDF5 files.

Public Attributes

- integer(hid_t) hdf5_i
  HDF5 file handle.
- integer xdmf_i
  XDMF file handle.
- character(len=max_str_ln) name
  name of files (without extensions ".h5" and ".xmf")

C.42.1 Detailed Description

HDF5 data type, containing the information about HDF5 files.

Definition at line 40 of file HDF5_vars.f90.

C.42.2 Member Data Documentation

C.42.2.1 hdf5_i

```
integer(hid_t) hdf5_vars::hdf5_file_type::hdf5_i
```

HDF5 file handle.

Definition at line 41 of file HDF5_vars.f90.

C.42.2.2 name

```
character(len=max_str_ln) hdf5_vars::hdf5_file_type::name
```

name of files (without extensions ".h5" and ".xmf")

Definition at line 43 of file HDF5_vars.f90.

C.42.2.3 xdmf_i

```
integer hdf5_vars::hdf5_file_type::xdmf_i
```

XDMF file handle.

Definition at line 42 of file HDF5_vars.f90.

C.43 mpi_vars::lock_type Type Reference

lock type

Public Member Functions

- procedure init (lock_loc, wu_tag)
  initialize
- procedure dealloc (lock_loc)
  deallocate

Public Attributes

- integer, dimension(:), allocatable wl
  waiting list
- integer wl_win
  window to waiting list
- integer wu_tag
  wakeup tag
- logical blocking
  is a normal blocking process

C.43.1 Detailed Description

lock type

There is a blocking (BL) and a nonblocking (NB) version where the former requires an exclusive lock and the latter a shared one. This is saved in the variable `blocking`.

NB processes that get the lock directly on request (meaning that there were no other processes in the queue) notify directly all the next NB processes after gaining access. It also sets their status to active. When a NB process gains the lock when notified after waiting, it does not have to check for other NB processes, as this has been done by the notifying process.

A BL process retains exclusive access upon receipt of the lock. Similarly to NB processes, if the receipt was direct on request, the status is set to active, but only of this NB process.

When returning the lock, all BL processes and NB that find themselves to be the last active NB process, scan the waiting list and pass the lock preferably to another BL process to notify. If not available, it searches for all the NB processes to notify together.

The advantage of prefering BL processes after finishing a process is that this way NB processes are accumulated, and then quickly finished afterwards.

Note

Every process in the waiting queue will eventually receive a notification.

Scheme:

	request access	**gain acces**	**return access**
BL	• add to queue • wait	if direct: • activate	• remove from queue • find next BL/NB • notify • activate all
NB	• add to queue • wait	if direct: • find next NB for all next NB: • notify • activate	always: • remove from queue if last NB: • find next BL/NB • notify • activate all

with preference BL > NB(s).

See also

This is based on an extension of the ideas in `RossAtomicIO`.

Definition at line 63 of file MPI_vars.f90.

C.43.2 Member Function/Subroutine Documentation

C.43.2.1 dealloc()

```
procedure mpi_vars::lock_type::dealloc (
              class(lock_type), intent(inout) lock_loc)
```

deallocate

deallocate

Definition at line 72 of file MPI_vars.f90.

C.43.2.2 init()

```
procedure mpi_vars::lock_type::init (
              class(lock_type), intent(inout) lock_loc,
              integer, intent(in) wu_tag)
```

initialize

initialize

Definition at line 70 of file MPI_vars.f90.

C.43.3 Member Data Documentation

C.43.3.1 blocking

```
logical mpi_vars::lock_type::blocking
```

is a normal blocking process

Definition at line 67 of file MPI_vars.f90.

C.43.3.2 wl

```
integer, dimension(:), allocatable mpi_vars::lock_type::wl
```

waiting list

Definition at line 64 of file MPI_vars.f90.

C.43.3.3 wl_win

```
integer mpi_vars::lock_type::wl_win
```

window to waiting list

Definition at line 65 of file MPI_vars.f90.

C.43.3.4 wu_tag

```
integer mpi_vars::lock_type::wu_tag
```

wakeup tag

Definition at line 66 of file MPI_vars.f90.

C.44 x_vars::modes_type Type Reference

mode number type

Public Member Functions

- procedure dealloc (mds)
  deallocate

Public Attributes

- integer, dimension(:,:), allocatable n
  n for all modes, in total grid
- integer, dimension(:,:), allocatable m
  m for all modes, in total grid
- integer, dimension(:,:), allocatable sec
  m or n for all possible modes, index and limits, in total grid

C.44.1 Detailed Description

mode number type

Type containing information about mode numbers at every flux surface:

- mode numbers: n, m,
- information of secondary modes: sec,
  - 1: mode number
  - 2: lower limit of normal range
  - 3: upper limit of normal range
  - 4: index in tables where the number of modes can be bigger than the range if the safety factor or rotational transform is non-monotonous so that the same mode number can have multiple ranges.
- Flux normal coordinate: r_F.

These are set up in setup_nm_x().

Definition at line 36 of file X_vars.f90.

C.44.2 Member Function/Subroutine Documentation

C.44.2.1 dealloc()

```
procedure x_vars::modes_type::dealloc (
              class(modes_type), intent(inout) mds)
```

deallocate

Parameters

in,out	*mds*	modes variables to be deallocated

Definition at line 42 of file X_vars.f90.

C.44.3 Member Data Documentation

C.44.3.1 m

```
integer, dimension(:,:), allocatable x_vars::modes_type::m
```

m for all modes, in total grid

Definition at line 38 of file X_vars.f90.

C.44.3.2 n

```
integer, dimension(:,:), allocatable x_vars::modes_type::n
```

n for all modes, in total grid

Definition at line 37 of file X_vars.f90.

C.44.3.3 sec

```
integer, dimension(:,:), allocatable x_vars::modes_type::sec
```

m or n for all possible modes, index and limits, in total grid

Definition at line 39 of file X_vars.f90.

C.45 num_utilities::order_per_fun Interface Reference

Order a periodic function to include $0 \ldots 2\pi$ and an overlap.

Public Member Functions

- integer function order_per_fun_1 (x, y, x_out, y_out, overlap, tol)
  version with x and y separate
- integer function order_per_fun_2 (xy, xy_out, overlap, tol)
  version with x and y together

C.45.1 Detailed Description

Order a periodic function to include $0 \ldots 2\pi$ and an overlap.

Returns

ierr

Definition at line 248 of file num_utilities.f90.

C.45.2 Member Function/Subroutine Documentation

C.45.2.1 order_per_fun_1()

```
integer function num_utilities::order_per_fun::order_per_fun_1 (
                real(dp), dimension(:), intent(in) x,
                real(dp), dimension(:), intent(in) y,
                real(dp), dimension(:), intent(inout), allocatable x_out,
                real(dp), dimension(:), intent(inout), allocatable y_out,
                integer, intent(in) overlap,
                real(dp), intent(in), optional tol)
```

version with x and y separate

Parameters

in	*x*	abscissa
in	*y*	ordinate
in,out	*x_out*	ordered x
in,out	*y_out*	ordered y
in	*overlap*	overlap to include
in	*tol*	tolerance for error

Definition at line 1573 of file num_utilities.f90.

C.45.2.2 order_per_fun_2()

```
integer function num_utilities::order_per_fun::order_per_fun_2 (
                real(dp), dimension(:,:), intent(in) xy,
                real(dp), dimension(:,:), intent(inout), allocatable xy_out,
                integer, intent(in) overlap,
                real(dp), intent(in), optional tol)
```

version with x and y together

Parameters

in	*xy*	abscissa and ordinate

in,out	*xy_out*	ordered xy
in	*overlap*	overlap to include
in	*tol*	tolerance for error

Definition at line 1649 of file num_utilities.f90.

C.46 output_ops::plot_hdf5 Interface Reference

Prints variables `vars` with names `var_names` in an HDF5 file with name `c` file_name and accompanying XDMF file.

Public Member Functions

- subroutine plot_hdf5_ind (var_name, file_name, var, tot_dim, loc_offset, x, y, z, cont_plot, descr)

  individual version
- subroutine plot_hdf5_arr (var_names, file_name, vars, tot_dim, loc_offset, x, y, z, col_id, col, sym_type, cont_plot, descr)

  array version

C.46.1 Detailed Description

Prints variables `vars` with names `var_names` in an HDF5 file with name `c` file_name and accompanying XDMF file.

For XDMF collections ([18]), only the first value for `var_names` is used, so it should have a size of one.

The plot is generally 3-D, but if one of the dimensions provided is equal to 1, it is checked whether there is poloidal or toroidal axisymmetry and if so, the plot becomes 2-D. This can be forced using the optional input argument `sym_type`.

Optionally, the (curvilinear) grid can be provided through `X`, `Y` and `Z`. If not, the grid is assumed to be Cartesian with discrete indices where `X` corresponds to the first dimensions, `Y` to the second and `Z` to the third.

Additionally, the total grid size and local offset can be provided in `tot_dim` and `loc_offset` to run this routine in parallel automatically. Optionally, one of the dimensions (`col_id`, default 4) can be associated to a collection dimension using `col` different from 1:

- `col` = 1: no collection, just plots of different variables
- `col` = 2: time collection
- `col` = 3: spatial collection
- `col` = 4: vector field

Furthermore, using the variable `cont_plot`, a plot can be (over-)written in multiple writes. By this is meant that there should be an initial plot, with collection type 1, 2, 3 or 4, which can then be followed by an arbitrary number of additional writes. As these additional writes currently cannot modify the plot structure, nor the XDMF variables, their collection dimension should be complete from the start.

This has no implications for single plots but means that for collection types 2 to 4 all the elements in the collection have to be present, though they do not necessary need to have been completely written in the other dimensions.

Subsequent writes with `cont_plot` can then, for instance, write parts of the data that had not yet been written, or overwrite ones that had. This can be useful for post-processing where the memory requirements are large so that the work has to be split.

Note

1. For a vector field, the number of variables has to be 2 for 2-D plots or 3 for 3-D plots. This should be rewritten in the future, so that collections can be used for vectors as well.
2. In order to merge collections in their collection dimension, the XDMF files can always easily be joined.
3. If necessary, a lock system should be used when multiple processes are writing the same file, including continued writes.
4. To plot this with VisIt, use:
   - for temporal collections: pseudocolor using the variable name (other names are ignored).
   - for spatial collections: subset of blocks or pseudocolor using the variable name (other names are ignored).
   - for vector plot: Vector plot.
   - for without collections: pseudocolor using the variable names.
5. Currently all of possibly multiple processes that write data have to cover the entire range of the variables in the dimension indicated by `col_id`. (This could be implemented by changing how n↩_plot is defined and selectively letting each processer write in the main loop at its corresponding indices.)
6. To project the data to 2-D in VisIt, use the projection tool under Operators > Transform

Definition at line 137 of file output_ops.f90.

C.46.2 Member Function/Subroutine Documentation

C.46.2.1 plot_hdf5_arr()

```
subroutine output_ops::plot_hdf5::plot_hdf5_arr (
            character(len=*), dimension(:), intent(in) var_names,
            character(len=*), intent(in) file_name,
            real(dp), dimension(:,:,:,:), intent(in), target vars,
            integer, dimension(4), intent(in), optional tot_dim,
            integer, dimension(4), intent(in), optional loc_offset,
            real(dp), dimension(:,:,:,:), intent(in), optional, target x,
            real(dp), dimension(:,:,:,:), intent(in), optional, target y,
            real(dp), dimension(:,:,:,:), intent(in), optional, target z,
            integer, intent(in), optional col_id,
            integer, intent(in), optional col,
            integer, intent(in), optional sym_type,
```

```
logical, intent(in), optional cont_plot,
character(len=*), intent(in), optional descr)
```

array version

Parameters

in	*var_names*	names of variable to be plot

in	*file_name*	file name
in	*vars*	variables to plot
in	*tot_dim*	total dimensions of the arrays
in	*loc_offset*	offset of local dimensions
in	*x*	curvilinear grid X points
in	*y*	curvilinear grid Y points
in	*z*	curvilinear grid Z points
in	*col_id*	index of time dimension
in	*col*	whether a collection is made
in	*sym_type*	type of symmetry (1: no symmetry, 2: toroidal, 3: poloidal)
in	*cont_plot*	continued plot
in	*descr*	description

Definition at line 433 of file output_ops.f90.

Here is the call graph for this function:

C.46.2.2 plot_hdf5_ind()

```
subroutine output_ops::plot_hdf5::plot_hdf5_ind (
            character(len=*), intent(in) var_name,
            character(len=*), intent(in) file_name,
            real(dp), dimension(:,:,:), intent(in) var,
            integer, dimension(3), intent(in), optional tot_dim,
            integer, dimension(3), intent(in), optional loc_offset,
            real(dp), dimension(:,:,:), intent(in), optional, target x,
```

```
real(dp), dimension(:,:,:), intent(in), optional, target y,
real(dp), dimension(:,:,:), intent(in), optional, target z,
logical, intent(in), optional cont_plot,
character(len=*), intent(in), optional descr)
```

individual version

Parameters

in	*var_name*	name of variable to be plot
in	*file_name*	file name
in	*var*	variable to plot
in	*tot_dim*	total dimensions of the arrays
in	*loc_offset*	offset of local dimensions
in	*x*	curvilinear grid X points
in	*y*	curvilinear grid Y points
in	*z*	curvilinear grid Z points
in	*cont_plot*	continued plot
in	*descr*	description

Definition at line 947 of file output_ops.f90.

C.47 output_ops::print_ex_2d Interface Reference

Print 2-D output on a file.

Public Member Functions

- subroutine print_ex_2d_ind (var_name, file_name_i, y, x, draw, persistent)
  individual version
- subroutine print_ex_2d_arr (var_names, file_name_i, y, x, draw, persistent)
  array version

C.47.1 Detailed Description

Print 2-D output on a file.

The variables `var_name` and `file_name` hold the name of the plot and of the file in which the plot data is to be saved, respectively. `y` is the an array containing the function which is stored and `x` is an optional vector with the x-values. The logical `draw` can optionally disable calling the external drawing procedure for output on screen [default], without modifying the plot file.

The first index of y (and x) contains the points of a current plot.

The second index indicates various plots (one or more)

Definition at line 47 of file output_ops.f90.

C.47.2 Member Function/Subroutine Documentation

C.47.2.1 print_ex_2d_arr()

```
subroutine output_ops::print_ex_2d::print_ex_2d_arr (
            character(len=*), dimension(:), intent(in) var_names,
            character(len=*), intent(in) file_name_i,
            real(dp), dimension(1:,1:), intent(in) y,
            real(dp), dimension(1:,1:), intent(in), optional x,
            logical, intent(in), optional draw,
            logical, intent(in), optional persistent)
```

array version

Parameters

in	*var_names*	names of variables in legend
in	*file_name_i*	name of input file
in	*y*	ordinate
in	*x*	absicca
in	*draw*	whether to draw the plot as well
in	*persistent*	keep on-screen plot open

Definition at line 171 of file output_ops.f90.

Here is the call graph for this function:

C.47.2.2 print_ex_2d_ind()

```
subroutine output_ops::print_ex_2d::print_ex_2d_ind (
            character(len=*), intent(in) var_name,
            character(len=*), intent(in) file_name_i,
```

```
real(dp), dimension(1:), intent(in) y,
real(dp), dimension(1:), intent(in), optional x,
logical, intent(in), optional draw,
logical, intent(in), optional persistent)
```

individual version

Parameters

in	*var_name*	name of variable in legend

in	*file_name_i*	name of input file
in	*y*	ordinate
in	*x*	absicca
in	*draw*	whether to draw the plot as well
in	*persistent*	keep on-screen plot open

Definition at line 146 of file output_ops.f90.

C.48 output_ops::print_ex_3d Interface Reference

Print 3-D output on a file.

Public Member Functions

- subroutine print_ex_3d_ind (var_name, file_name_i, z, y, x, draw)
  individual version
- subroutine print_ex_3d_arr (var_names, file_name_i, z, x, y, draw)
  array version

C.48.1 Detailed Description

Print 3-D output on a file.

The variables `var_name` and `file_name` hold the name of the plot and of the file in which the plot data is to be saved, respectively. z is the an array containing the function which is stored and x andy are optional vectors with the x and y-values. The logical `draw` can optionally disable calling the external drawing procedure for output on screen [default], without modifying the plot file.

The first index of z (and x, y) contains the points of a current The plot second index indicates various plots (one or more)

Definition at line 65 of file output_ops.f90.

C.48.2 Member Function/Subroutine Documentation

C.48.2.1 print_ex_3d_arr()

```
subroutine output_ops::print_ex_3d::print_ex_3d_arr (
            character(len=*), dimension(:), intent(in) var_names,
            character(len=*), intent(in) file_name_i,
            real(dp), dimension(1:,1:,1:), intent(in) z,
            real(dp), dimension(1:,1:,1:), intent(in), optional x,
            real(dp), dimension(1:,1:,1:), intent(in), optional y,
            logical, intent(in), optional draw)
```

array version

Parameters

in	*var_names*	names of variables in legend

in	*file_name_i*	name of input file
in	*z*	ordinate
in	*x*	absicca
in	*y*	absicca
in	*draw*	whether to draw the plot as well

Definition at line 309 of file output_ops.f90.

Here is the call graph for this function:

C.48.2.2 print_ex_3d_ind()

```
subroutine output_ops::print_ex_3d::print_ex_3d_ind (
                character(len=*), intent(in) var_name,
                character(len=*), intent(in) file_name_i,
                real(dp), dimension(1:,1:), intent(in) z,
                real(dp), dimension(1:,1:), intent(in), optional y,
                real(dp), dimension(1:,1:), intent(in), optional x,
                logical, intent(in), optional draw)
```

individual version

Parameters

in	*var_name*	name of variable in legend
in	*file_name_i*	name of input file
in	*z*	ordinate
in	*x*	absicca
in	*y*	absicca
in	*draw*	whether to draw the plot as well

Definition at line 268 of file output_ops.f90.

C.49 eq_ops::print_output_eq Interface Reference

Print equilibrium quantities to an output file:

Public Member Functions

- integer function print_output_eq_1 (grid_eq, eq, data_name)
  flux version
- integer function print_output_eq_2 (grid_eq, eq, data_name, rich_lvl, par_div, dealloc_vars)
  metric version

C.49.1 Detailed Description

Print equilibrium quantities to an output file:

- flux:
  - `pres_FD`,
  - `q_saf_FD`,
  - `rot_t_FD`,
  - `flux_p_FD`,
  - `flux_t_FD`,
  - `rho`,
  - `S`,
  - `kappa_n`,
  - `kappa_g`,
  - `sigma`
- metric:
  - `g_FD`,
  - `h_FD`,
  - `jac_FD`

If `rich_lvl` is provided, `"_R_[rich_lvl]"` is appended to the data name if it is $>$`0` (only for `eq_2`).

Optionally, for `eq_2`, it can be specified that this is a divided parallel grid, corresponding to the variable `eq_jobs_lims` with index `eq_job_nr`. In this case, the total grid size is adjusted to the one specified by `eq_jobs_lims` and the grid is written as a subset.

Note

1. The equilibrium quantities are outputted in Flux coordinates.
2. The metric equilibrium quantities can be deallocated on the fly, which is useful if this routine is followed by a deallocation any way, so that memory usage does not almost double.
3. `print_output_eq_2` is only used by HELENA now, as for VMEC it is too slow since there are often multiple VMEC equilibrium jobs, while for HELENA this is explicitely forbidden.

Returns

ierr

Definition at line 90 of file eq_ops.f90.

C.49.2 Member Function/Subroutine Documentation

C.49.2.1 print_output_eq_1()

```
integer function eq_ops::print_output_eq::print_output_eq_1 (
            type(grid_type), intent(in) grid_eq,
            type(eq_1_type), intent(in) eq,
            character(len=*), intent(in) data_name)
```

flux version

Parameters

in	*grid_eq*	equilibrium grid variables
in	*eq*	flux equilibrium variables
in	*data_name*	name under which to store

Definition at line 2274 of file eq_ops.f90.

Here is the call graph for this function:

C.49.2.2 print_output_eq_2()

```
integer function eq_ops::print_output_eq::print_output_eq_2 (
            type(grid_type), intent(in) grid_eq,
            type(eq_2_type), intent(inout) eq,
            character(len=*), intent(in) data_name,
            integer, intent(in), optional rich_lvl,
            logical, intent(in), optional par_div,
            logical, intent(in), optional dealloc_vars)
```

metric version

Parameters

in	*grid_eq*	equilibrium grid variables

in,out	*eq*	metric equilibrium variables
in	*data_name*	name under which to store
in	*rich_lvl*	Richardson level to print
in	*par_div*	is a parallely divided grid
in	*dealloc_vars*	deallocate variables on the fly after writing

Definition at line 2406 of file eq_ops.f90.

Here is the call graph for this function:

C.50 x_ops::print_output_x Interface Reference

Print either vectorial or tensorial perturbation quantities of a certain order to an output file.

Public Member Functions

- integer function print_output_x_1 (grid_x, x, data_name, rich_lvl, par_div, lim_sec_x)
  vectorial version.
- integer function print_output_x_2 (grid_x, x, data_name, rich_lvl, par_div, lim_sec_x, is_field_averaged)
  tensorial version

C.50.1 Detailed Description

Print either vectorial or tensorial perturbation quantities of a certain order to an output file.

- vectorial:
  - U
  - DU
- tensorial:
  - PV_int

- KV_int

(The non-integrated variables are not saved because they are heavy and not requested.)

If `rich_lvl` is provided, `"_R_[rich_lvl]"` is appended to the data name if it is $>$`0`.

Optionally, it can be specified that this is a divided parallel grid, corresponding to the variable `eq_jobs_lims` with index `eq_job_nr`. In this case, the total grid size is adjusted to the one specified by `eq_jobs_lims` and the grid is written as a subset.

Note

1. The equilibrium quantities are outputted in Flux coordinates.
2. The tensorial perturbation type can also be used for field-aligned variables, in which case the first index is assumed to have dimension 1 only. This can be triggered using `is_field_averaged`.

Returns

ierr

Definition at line 85 of file X_ops.f90.

C.50.2 Member Function/Subroutine Documentation

C.50.2.1 print_output_x_1()

```
integer function x_ops::print_output_x::print_output_x_1 (
                type(grid_type), intent(in) grid_x,
                type(x_1_type), intent(in) x,
                character(len=*), intent(in) data_name,
                integer, intent(in), optional rich_lvl,
                logical, intent(in), optional par_div,
                integer, dimension(2), intent(in), optional lim_sec_x)
```

vectorial version.

Parameters

in	*grid_x*	perturbation grid variables
in	*x*	vectorial perturbation variables
in	*data_name*	name under which to store
in	*rich_lvl*	Richardson level to print
in	*par_div*	is a parallely divided grid

in	*lim_sec_x*	limits of m_X (pol. flux) or n_X (tor. flux)

Definition at line 409 of file X_ops.f90.

Here is the call graph for this function:

C.50.2.2 print_output_x_2()

```
integer function x_ops::print_output_x::print_output_x_2 (
                type(grid_type), intent(in) grid_x,
                type(x_2_type), intent(in) x,
                character(len=*), intent(in) data_name,
                integer, intent(in), optional rich_lvl,
                logical, intent(in), optional par_div,
                integer, dimension(2,2), intent(in), optional lim_sec_x,
                logical, intent(in), optional is_field_averaged)
```

tensorial version

Parameters

in	*grid_x*	perturbation grid variables
in	*x*	tensorial perturbation variables
in	*data_name*	name under which to store
in	*rich_lvl*	Richardson level to print
in	*par_div*	is a parallely divided grid
in	*lim_sec_x*	limits of m_X (pol. flux) or n_X (tor. flux)
in	*is_field_averaged*	whether field-averaged, so only one dimension for first index

Definition at line 601 of file X_ops.f90.

Here is the call graph for this function:

C.51 eq_ops::redistribute_output_eq Interface Reference

Redistribute the equilibrium variables, but only the Flux variables are saved.

Public Member Functions

- integer function redistribute_output_eq_1 (grid, grid_out, eq, eq_out)
  flux version
- integer function redistribute_output_eq_2 (grid, grid_out, eq, eq_out)
  metric version

C.51.1 Detailed Description

Redistribute the equilibrium variables, but only the Flux variables are saved.

See also

redistribute_output_grid()

Returns

ierr

Definition at line 103 of file eq_ops.f90.

C.51.2 Member Function/Subroutine Documentation

C.51.2.1 redistribute_output_eq_1()

```
integer function eq_ops::redistribute_output_eq::redistribute_output_eq_1 (
                type(grid_type), intent(in) grid,
                type(grid_type), intent(in) grid_out,
                type(eq_1_type), intent(in) eq,
                type(eq_1_type), intent(inout) eq_out)
```

flux version

Parameters

in	*grid*	equilibrium grid variables

in	*grid_out*	redistributed equilibrium grid variables
in	*eq*	flux equilibrium variables
in,out	*eq_out*	flux equilibrium variables in redistributed grid

Definition at line 2595 of file eq_ops.f90.

Here is the call graph for this function:

C.51.2.2 redistribute_output_eq_2()

```
integer function eq_ops::redistribute_output_eq::redistribute_output_eq_2 (
            type(grid_type), intent(in) grid,
            type(grid_type), intent(in) grid_out,
            type(eq_2_type), intent(in) eq,
            type(eq_2_type), intent(inout) eq_out)
```

metric version

Parameters

in	*grid*	equilibrium grid variables
in	*grid_out*	redistributed equilibrium grid variables
in	*eq*	metric equilibrium variables
in,out	*eq_out*	metric equilibrium variables in redistributed grid

Definition at line 2663 of file eq_ops.f90.

Here is the call graph for this function:

C.52 x_ops::redistribute_output_x Interface Reference

Redistribute the perturbation variables.

Public Member Functions

- integer function redistribute_output_x_1 (mds, grid, grid_out, x, x_out)
  flux version
- integer function redistribute_output_x_2 (mds, grid, grid_out, x, x_out)
  metric version

C.52.1 Detailed Description

Redistribute the perturbation variables.

See also

redistribute_output_grid()

Returns

ierr

Definition at line 51 of file X_ops.f90.

C.52.2 Member Function/Subroutine Documentation

C.52.2.1 redistribute_output_x_1()

```
integer function x_ops::redistribute_output_x::redistribute_output_x_1 (
                type(modes_type), intent(in) mds,
                type(grid_type), intent(in) grid,
                type(grid_type), intent(in) grid_out,
                type(x_1_type), intent(in) x,
                type(x_1_type), intent(inout) x_out)
```

flux version

Parameters

in	*mds*	general modes variables
in	*grid*	perturbation grid variables
in	*grid_out*	redistributed perturbation grid variables
in	*x*	vectorial perturbation variables

in,out	*x_out*	vectorial perturbation variables in redistributed grid

Definition at line 176 of file X_ops.f90.

Here is the call graph for this function:

C.52.2.2 redistribute_output_x_2()

```
integer function x_ops::redistribute_output_x::redistribute_output_x_2 (
                type(modes_type), intent(in) mds,
                type(grid_type), intent(in) grid,
                type(grid_type), intent(in) grid_out,
                type(x_2_type), intent(in) x,
                type(x_2_type), intent(inout) x_out)
```

metric version

Parameters

in	*mds*	general modes variables
in	*grid*	perturbation grid variables
in	*grid_out*	redistributed perturbation grid variables
in	*x*	tensorial perturbation variables
in,out	*x_out*	tensorial perturbation variables in redistributed grid

Definition at line 275 of file X_ops.f90.

Here is the call graph for this function:

C.53 hdf5_ops::reset_hdf5_item Interface Reference

Resets an HDF5 XDMF item.

Public Member Functions

- subroutine reset_hdf5_item_ind (xdmf_item, ind_plot)

  individual version
- subroutine reset_hdf5_item_arr (xdmf_items, ind_plot)

  array version

C.53.1 Detailed Description

Resets an HDF5 XDMF item.

Note

individual version cannot make use of array version because then the deallocation does not work properly.

Definition at line 57 of file HDF5_ops.f90.

C.53.2 Member Function/Subroutine Documentation

C.53.2.1 reset_hdf5_item_arr()

```
subroutine hdf5_ops::reset_hdf5_item::reset_hdf5_item_arr (
            type(xml_str_type), dimension(:), intent(inout) xdmf_items,
            logical, intent(in), optional ind_plot)
```

array version

Parameters

in,out	*xdmf_items*	XDMF items to reset
in	*ind_plot*	whether individual plot

Definition at line 1793 of file HDF5_ops.f90.

Here is the call graph for this function:

C.53.2.2 reset_hdf5_item_ind()

```
subroutine hdf5_ops::reset_hdf5_item::reset_hdf5_item_ind (
                type(xml_str_type), intent(inout) xdmf_item,
                logical, intent(in), optional ind_plot)
```

individual version

Parameters

in,out	*xdmf_item*	XDMF item to reset

in	*ind_plot*	whether individual plot

Definition at line 1833 of file HDF5_ops.f90.

Here is the call graph for this function:

C.54 num_utilities::round_with_tol Interface Reference

Rounds an arry of values to limits, with a tolerance 10^{-5} that can optionally be modified.

Public Member Functions

- integer function round_with_tol_ind (val, lim_lo, lim_hi, tol)
  individual version
- integer function round_with_tol_arr (vals, lim_lo, lim_hi, tol)
  array version

C.54.1 Detailed Description

Rounds an arry of values to limits, with a tolerance 10^{-5} that can optionally be modified.

Definition at line 169 of file num_utilities.f90.

C.54.2 Member Function/Subroutine Documentation

C.54.2.1 round_with_tol_arr()

```
integer function num_utilities::round_with_tol::round_with_tol_arr (
            real(dp), dimension(:), intent(inout) vals,
            real(dp), intent(in) lim_lo,
            real(dp), intent(in) lim_hi,
            real(dp), intent(in), optional tol)
```

array version

Parameters

in,out	*vals*	values to be rounded

in	*lim_lo*	lower limit on val
in	*lim_↩ hi*	upper limit on val
in	*tol*	tolerance

Definition at line 1382 of file num_utilities.f90.

C.54.2.2 round_with_tol_ind()

```
integer function num_utilities::round_with_tol::round_with_tol_ind (
            real(dp), intent(inout) val,
            real(dp), intent(in) lim_lo,
            real(dp), intent(in) lim_hi,
            real(dp), intent(in), optional tol)
```

individual version

Parameters

in,out	*val*	value to be rounded
in	*lim_lo*	lower limit on val
in	*lim_↩ hi*	upper limit on val
in	*tol*	tolerance

Definition at line 1420 of file num_utilities.f90.

C.55 x_utilities::sec_ind_loc2tot Interface Reference

Returns the `sec_ind_tot` used to refer to a perturbation quantity.

Public Member Functions

- integer function sec_ind_loc2tot_1 (id, lim_sec_x)
  vectorial version
- integer function, dimension(2) sec_ind_loc2tot_2 (id, jd, lim_sec_x)
  tensorial version

C.55.1 Detailed Description

Returns the `sec_ind_tot` used to refer to a perturbation quantity.

Definition at line 22 of file X_utilities.f90.

C.55.2 Member Function/Subroutine Documentation

C.55.2.1 sec_ind_loc2tot_1()

```
integer function x_utilities::sec_ind_loc2tot::sec_ind_loc2tot_1 (
                integer, intent(in) id,
                integer, dimension(2), intent(in), optional lim_sec_x)
```

vectorial version

Parameters

in	*id*	mode index
in	*lim_sec↩ _x*	limits of m_X (pol. flux) or n_X (tor. flux)

Returns

output

Definition at line 31 of file X_utilities.f90.

C.55.2.2 sec_ind_loc2tot_2()

```
integer function, dimension(2) x_utilities::sec_ind_loc2tot::sec_ind_loc2tot_2 (
                integer, intent(in) id,
                integer, intent(in) jd,
                integer, dimension(2,2), intent(in), optional lim_sec_x)
```

tensorial version

Parameters

in	*id*	mode index for dimension 1
in	*jd*	mode index for dimension 1
in	*lim_sec↩ _x*	limits of m_X (pol flux) or n_X (tor flux) for both dimensions

Returns

output

Definition at line 48 of file X_utilities.f90.

C.56 x_vars::set_nm_x Interface Reference

Sets n_X and m_X.

Public Member Functions

- subroutine set_nm_x_1 (mds, grid_x, lim_sec_x_o, n_x_loc, m_x_loc, lim_sec_x_i)
  vectorial version
- subroutine set_nm_x_2 (mds, grid_x, lim_sec_x_o, n_x_1, m_x_1, n_x_2, m_x_2, lim_sec_x_i)
  tensorial version

C.56.1 Detailed Description

Sets `n_X` and `m_X`.

By default, this is done using by default global `X_vars` variables but optionally different limits for the secondary mode numbers (`m_X` for poloidal flux or `n_X` for toroidal flux).

Note

`n_X` and `m_X` need to have been set up with the same limits as the grid used here. This is done in setup↩_nm_X().

Definition at line 116 of file X_vars.f90.

C.56.2 Member Function/Subroutine Documentation

C.56.2.1 set_nm_x_1()

```
subroutine x_vars::set_nm_x::set_nm_x_1 (
            type(modes_type), intent(in) mds,
            type(grid_type), intent(in) grid_x,
            integer, dimension(2), intent(inout) lim_sec_x_o,
            integer, dimension(:,:), intent(inout), allocatable n_x_loc,
            integer, dimension(:,:), intent(inout), allocatable m_x_loc,
            integer, dimension(2), intent(in), optional lim_sec_x_i)
```

vectorial version

Parameters

in	*mds*	general modes variables
in	*grid_x*	perturbation grid
in,out	*lim_sec_x↩_o*	limits on secondary mode numbers
in,out	*n_x_loc*	toroidal mode numbers
in,out	*m_x_loc*	poloidal mode numbers
in	*lim_sec_x↩_i*	optional limits on secondary mode numbers

Definition at line 144 of file X_vars.f90.

C.56.2.2 set_nm_x_2()

```
subroutine x_vars::set_nm_x::set_nm_x_2 (
                type(modes_type), intent(in) mds,
                type(grid_type), intent(in) grid_x,
                integer, dimension(2,2), intent(inout) lim_sec_x_o,
                integer, dimension(:,:), intent(inout), allocatable n_x_1,
                integer, dimension(:,:), intent(inout), allocatable m_x_1,
                integer, dimension(:,:), intent(inout), allocatable n_x_2,
                integer, dimension(:,:), intent(inout), allocatable m_x_2,
                integer, dimension(2,2), intent(in), optional lim_sec_x_i)
```

tensorial version

Parameters

in	*mds*	general modes variables
in	*grid_x*	perturbation grid
in,out	*lim_sec_x↩_o*	limits on secondary mode numbers
in,out	*n_x_1*	toroidal mode numbers for dimension 1
in,out	*m_x_1*	poloidal mode numbers for dimension 1
in,out	*n_x_2*	toroidal mode numbers for dimension 2
in,out	*m_x_2*	poloidal mode numbers for dimension 2
in	*lim_sec_x↩_i*	optional limits on secondary mode numbers

Definition at line 167 of file X_vars.f90.

C.57 sol_vars::sol_type Type Reference

solution type

Public Member Functions

- procedure init (sol, mds, grid_sol, n_ev, lim_sec_x)

  initialize
- procedure dealloc (sol)

  deallocate

Public Attributes

- integer n_mod
  size of n and m (nr. of modes)
- integer, dimension(2) lim_sec_x
  limits of m_X (pol. flux) or n_X (tor. flux)
- integer, dimension(:,:), allocatable n
  vector of toroidal mode numbers
- integer, dimension(:,:), allocatable m
  vector of poloidal mode numbers
- complex(dp), dimension(:,:,:), allocatable vec
  Eigenvector solution.
- complex(dp), dimension(:), allocatable val
  Eigenvalue solution.
- real(dp) estim_mem_usage
  estimated memory usage

C.57.1 Detailed Description

solution type

The arrays here are of the form:

- `val: (1:n_EV)`
- `vec: (1:n_mod,1:loc_n_r,1:n_EV)`

Definition at line 30 of file sol_vars.f90.

C.57.2 Member Function/Subroutine Documentation

C.57.2.1 dealloc()

```
procedure sol_vars::sol_type::dealloc (
            class(sol_type), intent(inout) sol)
```

deallocate

Note

intent(out) automatically deallocates the variable

Parameters

in,out	*sol*	solution variables to be deallocated

Definition at line 44 of file sol_vars.f90.

C.57.2.2 init()

```
procedure sol_vars::sol_type::init (
                class(sol_type), intent(inout) sol,
                type(modes_type), intent(in) mds,
                type(grid_type), intent(in) grid_sol,
                integer, intent(in) n_ev,
                integer, dimension(2), intent(in), optional lim_sec_x)
```

initialize

The number of modes as well as `n` and `m` are also set up.

Optionally, the secondary mode number can be specified (`m` if poloidal flux is used as normal coordinate and c n if toroidal flux). By default, they are taken from the global `X_vars` variables.

Note

If the lowest limits of the grid is not 1 (e.g. `grid_soli_min = 1` for first process), the input variable `i_min` should be set to set correctly. For a full grid, it should be set to 1.

Parameters

`in,out`	*sol*	solution variables
`in`	*mds*	general modes variables
`in`	*grid_sol*	solution grid
`in`	*n_ev*	nr. of Eigenvalues
`in`	*lim_sec_x*	limits of `m_X` (pol. flux) or `n_X` (tor. flux)

Definition at line 42 of file sol_vars.f90.

C.57.3 Member Data Documentation

C.57.3.1 estim_mem_usage

```
real(dp) sol_vars::sol_type::estim_mem_usage
```

estimated memory usage

Note

Debug version only

Definition at line 38 of file sol_vars.f90.

C.57.3.2 lim_sec_x

```
integer, dimension(2) sol_vars::sol_type::lim_sec_x
```

limits of `m_X` (pol. flux) or `n_X` (tor. flux)

Definition at line 32 of file sol_vars.f90.

C.57.3.3 m

```
integer, dimension(:,:), allocatable sol_vars::sol_type::m
```

vector of poloidal mode numbers

Definition at line 34 of file sol_vars.f90.

C.57.3.4 n

```
integer, dimension(:,:), allocatable sol_vars::sol_type::n
```

vector of toroidal mode numbers

Definition at line 33 of file sol_vars.f90.

C.57.3.5 n_mod

```
integer sol_vars::sol_type::n_mod
```

size of n and m (nr. of modes)

Definition at line 31 of file sol_vars.f90.

C.57.3.6 val

```
complex(dp), dimension(:), allocatable sol_vars::sol_type::val
```

Eigenvalue solution.

Definition at line 36 of file sol_vars.f90.

C.57.3.7 vec

```
complex(dp), dimension(:,:,:), allocatable sol_vars::sol_type::vec
```

Eigenvector solution.

Definition at line 35 of file sol_vars.f90.

C.58 num_utilities::spline Interface Reference

Wrapper to the pspline library, making it easier to use for 1-D applications where speed is not the main priority. If spline representations are to be reused, manually use the library.

Public Member Functions

- integer function spline_real (x, y, xnew, ynew, ord, deriv, bcs, bcs_val, extrap)
  real version
- integer function spline_complex (x, y, xnew, ynew, ord, deriv, bcs, bcs_val, extrap)
  complex version

C.58.1 Detailed Description

Wrapper to the pspline library, making it easier to use for 1-D applications where speed is not the main priority. If spline representations are to be reused, manually use the library.

Order 1 (linear), 2 (akima hermite) or 3 (cubic) possible. Boundary conditions are possible:

- -1: periodic
- 0: not-a-knot
- 1: prescribe first derivative
- 2: prescribe second derivative

However, for order 2 boundary condition 2 is not available and for order 1 none of them.

Furthermore, derivatives can be specified:

- up to 1 for order 1 and 2
- up to 2 for order 3

Finally, extrapolation can be performed as well.

Returns

ierr

Definition at line 276 of file num_utilities.f90.

C.58.2 Member Function/Subroutine Documentation

C.58.2.1 spline_complex()

```
integer function num_utilities::spline::spline_complex (
                real(dp), dimension(:), intent(in) x,
                complex(dp), dimension(:), intent(in) y,
                real(dp), dimension(:), intent(in) xnew,
                complex(dp), dimension(:), intent(out) ynew,
                integer, intent(in), optional ord,
                integer, intent(in), optional deriv,
                integer, dimension(2), intent(in), optional bcs,
                complex(dp), dimension(2), intent(in), optional bcs_val,
                logical, intent(in), optional extrap)
```

complex version

Parameters

in	x	coordinates

in	*y*	function value
in	*xnew*	new coordinates
out	*ynew*	new function values
in	*ord*	order [def 3]
in	*deriv*	derivative [def 0]
in	*bcs*	boundary conditions [def 0]
in	*bcs_val*	boundary conditions [no def]
in	*extrap*	whether extrapolation is allowed [def .false.]

Definition at line 2001 of file num_utilities.f90.

C.58.2.2 spline_real()

```
integer function num_utilities::spline::spline_real (
                real(dp), dimension(:), intent(in), target x,
                real(dp), dimension(:), intent(in) y,
                real(dp), dimension(:), intent(in), target xnew,
                real(dp), dimension(:), intent(out) ynew,
                integer, intent(in), optional ord,
                integer, intent(in), optional deriv,
                integer, dimension(2), intent(in), optional bcs,
                real(dp), dimension(2), intent(in), optional bcs_val,
                logical, intent(in), optional extrap)
```

real version

Parameters

in	*x*	coordinates
in	*y*	function value
in	*xnew*	new coordinates
out	*ynew*	new function values
in	*ord*	order [def 3]
in	*deriv*	derivative [def 0]
in	*bcs*	boundary conditions [def 0]
in	*bcs_val*	boundary conditions [no def]
in	*extrap*	whether extrapolation is allowed [def .false.]

Definition at line 1674 of file num_utilities.f90.

Here is the call graph for this function:

C.59 eq_utilities::transf_deriv Interface Reference

Calculates derivatives in a coordinate system B from derivatives in a coordinates system A, making use of the transformation matrix $\overline{\mathsf{T}}^{\mathsf{A}}_{\mathsf{B}}$.

Public Member Functions

- integer recursive function transf_deriv_3_ind (x_a, t_ba, x_b, max_deriv, deriv_b, deriv_a_input)
  3-D scalar version with one derivative
- integer function transf_deriv_3_arr (x_a, t_ba, x_b, max_deriv, derivs)
  3-D scalar version with multiple derivatives
- integer function transf_deriv_3_arr_2d (x_a, t_ba, x_b, max_deriv, derivs)
  3-D matrix version with multiple derivatives
- integer recursive function transf_deriv_1_ind (x_a, t_ba, x_b, max_deriv, deriv_b, deriv_a_input)
  1-D scalar version with one derivative

C.59.1 Detailed Description

Calculates derivatives in a coordinate system B from derivatives in a coordinates system A, making use of the transformation matrix $\overline{\mathsf{T}}^{\mathsf{A}}_{\mathsf{B}}$.

The routine works by exchanging the derivatives in the coordinates B for derivatives in coordinates A using the formula

$$\mathbf{D}^m_{\mathsf{B}} X = \overline{\mathsf{T}}^{\mathsf{A}}_{\mathsf{B}} \mathbf{D}^1_{\mathsf{A}} \left(\mathbf{D}^{m-1}_{\mathsf{B}} X \right) ,$$

where $\mathbf{D}^m$ is a tensor of rank m that contains all the derivatives of total rank m. For example,

$$\mathbf{D}^1 = \vec{D} = \begin{pmatrix} \frac{\partial}{\partial y^1} \\ \frac{\partial}{\partial y^2} \\ \frac{\partial}{\partial u^3} \end{pmatrix}$$

This is done for the derivatives in each of the coordinates B until degree 0 is reached.

Furthermore, each of these degrees of derivatives in coordinates B can be be derived optionally in the original coordinate system A, which yields the formula:

$$\mathbf{D}_{\mathsf{A}}^{p}\left(\mathbf{D}_{\mathsf{B}}^{m}X\right)=\sum_{q}\left(\begin{array}{c}p\\q\end{array}\right)\mathbf{D}_{\mathsf{A}}^{q}\left(\overline{\mathsf{T}}_{\mathsf{B}}^{\mathsf{A}}\right)\left(\mathbf{D}_{\mathsf{A}}^{p-q}\mathbf{D}_{\mathsf{A}}^{1}\right)\left(\mathbf{D}_{\mathsf{B}}^{m-1}X\right)$$

This way, ultimately the desired derivatives in the coordinates B can be obtained recursively from the lower orders in the coordinates B and higher orders in the coordinates A.

For example:

- For order M, the formula can be used with $p=0$.
- For this, it is necessary that $\mathbf{D}_{\mathsf{A}}^{1}\mathbf{D}_{\mathsf{B}}^{m-1}X$ be precalculated, which can be done using the formula with $p=1$ and $m=M-1$.
- For this, it is necessary that $\mathbf{D}_{\mathsf{A}}^{1}\mathbf{D}_{\mathsf{B}}^{M-1}X$ and $\mathbf{D}_{\mathsf{A}}^{2}\mathbf{D}_{\mathsf{B}}^{M-1}X$ are precalculated, which can be done with $p=1$ and $m=M-1$ as well as $p=2$ and $m=M-1$.
- etc.

See also

[15] for more detailed information.

Returns

ierr

Definition at line 118 of file eq_utilities.f90.

C.59.2 Member Function/Subroutine Documentation

C.59.2.1 transf_deriv_1_ind()

```
integer recursive function eq_utilities::transf_deriv::transf_deriv_1_ind (
                real(dp), dimension(1:,0:), intent(in) x_a,
                real(dp), dimension(1:,0:), intent(in) t_ba,
                real(dp), dimension(1:), intent(inout) x_b,
                integer, intent(in) max_deriv,
                integer, intent(in) deriv_b,
                integer, intent(in), optional deriv_a_input)
```

1-D scalar version with one derivative

Parameters

in	*x_a*	variable and derivs. in coord. system A

in,out	*x_b*	requested derivs. of variable in coord. system B
in	*t_ba*	transf. mat. and derivs. between coord. systems A and B
in	*deriv_a_input*	derivs. in coord. system A (optional)
in	*deriv_b*	derivs. in coord. system B
in	*max_deriv*	maximum degrees of derivs.

Definition at line 679 of file eq_utilities.f90.

C.59.2.2 transf_deriv_3_arr()

```
integer function eq_utilities::transf_deriv::transf_deriv_3_arr (
                real(dp), dimension(1:,1:,1:,0:,0:,0:), intent(in) x_a,
                real(dp), dimension(1:,1:,1:,1:,0:,0:,0:), intent(in) t_ba,
                real(dp), dimension(1:,1:,1:,0:,0:,0:), intent(inout) x_b,
                integer, intent(in) max_deriv,
                integer, dimension(:,:), intent(in) derivs)
```

3-D scalar version with multiple derivatives

Parameters

in	*x_a*	variable and derivs. in coord. system A
in	*t_ba*	transf. mat. and derivs. between coord. systems A and B
in,out	*x_b*	requested derivs. of variable in coord. system B
in	*max_deriv*	maximum degrees of derivs.
in	*derivs*	series of derivs. (in coordinate system B)

Definition at line 613 of file eq_utilities.f90.

C.59.2.3 transf_deriv_3_arr_2d()

```
integer function eq_utilities::transf_deriv::transf_deriv_3_arr_2d (
                real(dp), dimension(1:,1:,1:,1:,0:,0:,0:), intent(in) x_a,
                real(dp), dimension(1:,1:,1:,1:,0:,0:,0:), intent(in) t_ba,
                real(dp), dimension(1:,1:,1:,1:,0:,0:,0:), intent(inout) x_b,
                integer, intent(in) max_deriv,
                integer, dimension(:,:), intent(in) derivs)
```

3-D matrix version with multiple derivatives

Parameters

in	*x_a*	variable and derivs. in coord. system A

in	*t_ba*	transf. mat. and derivs. between coord. systems A and B
in,out	*x_b*	requested derivs. of variable in coord. system B
in	*max_deriv*	maximum degrees of derivs.
in	*derivs*	series of derivs. (in coordinate system B)

Definition at line 638 of file eq_utilities.f90.

Here is the call graph for this function:

C.59.2.4 transf_deriv_3_ind()

```
integer recursive function eq_utilities::transf_deriv::transf_deriv_3_ind (
                real(dp), dimension(1:,1:,1:,0:,0:,0:), intent(in) x_a,
                real(dp), dimension(1:,1:,1:,1:,0:,0:,0:), intent(in) t_ba,
                real(dp), dimension(1:,1:,1:), intent(inout) x_b,
                integer, intent(in) max_deriv,
                integer, dimension(:), intent(in) deriv_b,
                integer, dimension(:), intent(in), optional deriv_a_input)
```

3-D scalar version with one derivative

Parameters

in	*x_a*	variable and derivs. in coord. system A
in	*t_ba*	transf. mat. and derivs. between coord. systems A and B
in,out	*x_b*	requested derivs. of variable in coord. system B
in	*max_deriv*	maximum degrees of derivs.

in	*deriv_b*	derivs. in coord. system B
in	*deriv_a_input*	derivs. in coord. system A (optional)

Definition at line 506 of file eq_utilities.f90.

C.60 vac_vars::vac_type Type Reference

vacuum type

Public Member Functions

- procedure init (vac, style, n_bnd, prim_x, n_ang, jq)
  initialize
- procedure dealloc (vac)
  deallocate

Public Attributes

- integer style
  style of vacuum (1: field-line 3-D, 2: axisymmetric)
- integer prim_x
  primary mode number
- integer ctxt_hg
  context for H and G
- integer n_bnd
  number of points in boundary
- integer bs
  block size in cyclical storage
- integer mpi_comm
  communicator for vacuum
- integer, dimension(blacsctxtsize) desc_h
  descriptor for H
- integer, dimension(blacsctxtsize) desc_g
  descriptor for G
- integer, dimension(2) n_p
  nr. of processes in grid
- integer, dimension(2) n_ang
  number of angles (1) and number of field lines (2)
- integer, dimension(2) ind_p
  index of local process in grid
- integer, dimension(2) n_loc
  local number of rows and columns
- integer, dimension(:), allocatable sec_x

secondary mode numbers

- integer, dimension(:,:), allocatable lims_c

  column limits for different subrows of G and H
- integer, dimension(:,:), allocatable lims_r

  row limits for different subcolumns of G and H
- real(dp) jq

  iota (tor. flux) or q (pol. flux) at edge
- real(dp), dimension(:,:), allocatable ang

  angle along field line, for each field line
- real(dp), dimension(:,:), allocatable norm

  J nabla psi normal vector.
- real(dp), dimension(:,:), allocatable dnorm

  poloidal derivative of norm (only for style 2)
- real(dp), dimension(:,:), allocatable h_fac

  metric factors (1,1), (1,3) and (3,3) (only for style 1)
- real(dp), dimension(:,:), allocatable x_vec

  Cartesian vector of position.
- real(dp), dimension(:,:), allocatable h

  H coefficient.
- real(dp), dimension(:,:), allocatable g

  G coefficient.
- complex(dp), dimension(:,:), allocatable res

  vacuum response
- real(dp) estim_mem_usage

  estimated memory usage

C.60.1 Detailed Description

vacuum type

The arrays here are of the form:

- `H, G: (n_loc,n_loc)`
- `res: (n_mod_X,n_mod_X)`

where `n_loc` is the number of points in the boundary, i.e. a subset of `n_bnd`.

For vacuum style 1, the vacuum is assumed to be equidsitant in the coordinate along the magnetic field lines and, if there are multiple field lines, also in the field line label α. The limits on the angles are assumed to be given by the global variables `grid_vars.min_par_x`, `grid_vars.max_par_x` and `grid_vars.min_alpha`, `grid_vars.max_alpha`.

For vacuum style 2, there is an additional angle `ang`. It is composed of the angles along the magnetic fields (which refers to `angle_1` in the discussion of `grid_vars.grid_type`), of which there can be multiple, but the total sum must be equal to `n_bnd`.

Definition at line 46 of file vac_vars.f90.

C.60.2 Member Function/Subroutine Documentation

C.60.2.1 dealloc()

```
procedure vac_vars::vac_type::dealloc (
                class(vac_type), intent(inout) vac)
```

deallocate

Parameters

in,out	*vac*	vacuum variables to be deallocated

Definition at line 78 of file vac_vars.f90.

C.60.2.2 init()

```
procedure vac_vars::vac_type::init (
                class(vac_type), intent(inout) vac,
                integer, intent(in) style,
                integer, intent(in) n_bnd,
                integer, intent(in) prim_x,
                integer, dimension(2), intent(in) n_ang,
                real(dp), intent(in) jq)
```

initialize

The number of modes as well as `n` and `m` are also set up.

The variables G and H are saved in a special format, based on the block-cyclical distribution employed in scalapack. As a hypothetical example, consider a process grid corresponding to block-size 1x2 and 2x3 processes with a total size of 3x15:

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 \\ 3 & 3 & 4 & 4 & 5 & 5 & 3 & 3 & 4 & 4 & 5 & 5 & 3 & 3 & 4 \\ 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 \end{bmatrix}$$

Therefore, the 3-D storage convention used is, for example, for process 1:

$$\begin{bmatrix} (1,3) & (1,4) & (1,9) & (1,10) & (1,15) & (1,\cdot) \\ (3,3) & (3,4) & (3,9) & (3,10) & (3,15) & (3,\cdot) \end{bmatrix}$$

The information concerning the delimitations of the individual subintervals in the horizontal and vertical direction can be saved once per subinterval. For process 4 of above example, this would be

$$\begin{bmatrix} 2 \\ 2 \end{bmatrix};$$

for the vertical direction, and

$$\begin{bmatrix} 3 & 9 & 15 \\ 4 & 10 & 15 \end{bmatrix};$$

for the horizontal direction.

A value $\cdot$ indicates that the index is out of global bounds. Practically, this is checked by seeing whether the total index does not exceed the global bounds.

Returns

ierr

Parameters

in,out	*vac*	vacuum variables

in	*style*	style of vacuum (1: field-line 3-D, 2: axisymmetric)
in	*n_bnd*	total number of points in boundary
in	*prim_x*	primary mode number
in	*n_ang*	number of angles (1) and number of field lines (2)
in	*jq*	iota (tor. flux) or q (pol. flux) at edge

Definition at line 76 of file vac_vars.f90.

C.60.3 Member Data Documentation

C.60.3.1 ang

```
real(dp), dimension(:,:), allocatable vac_vars::vac_type::ang
```

angle along field line, for each field line

Definition at line 63 of file vac_vars.f90.

C.60.3.2 bs

```
integer vac_vars::vac_type::bs
```

block size in cyclical storage

Definition at line 51 of file vac_vars.f90.

C.60.3.3 ctxt_hg

```
integer vac_vars::vac_type::ctxt_hg
```

context for H and G

Definition at line 49 of file vac_vars.f90.

C.60.3.4 desc_g

```
integer, dimension(blacsctxtsize) vac_vars::vac_type::desc_g
```

descriptor for G

Definition at line 54 of file vac_vars.f90.

C.60.3.5 desc_h

```
integer, dimension(blacsctxtsize) vac_vars::vac_type::desc_h
```

descriptor for H

Definition at line 53 of file vac_vars.f90.

C.60.3.6 dnorm

```
real(dp), dimension(:,:), allocatable vac_vars::vac_type::dnorm
```

poloidal derivative of norm (only for style 2)

Definition at line 65 of file vac_vars.f90.

C.60.3.7 estim_mem_usage

```
real(dp) vac_vars::vac_type::estim_mem_usage
```

estimated memory usage

Note

Debug version only

Definition at line 72 of file vac_vars.f90.

C.60.3.8 g

```
real(dp), dimension(:,:), allocatable vac_vars::vac_type::g
```

G coefficient.

Definition at line 69 of file vac_vars.f90.

C.60.3.9 h

```
real(dp), dimension(:,:), allocatable vac_vars::vac_type::h
```

H coefficient.

Definition at line 68 of file vac_vars.f90.

C.60.3.10 h_fac

```
real(dp), dimension(:,:), allocatable vac_vars::vac_type::h_fac
```

metric factors (1,1), (1,3) and (3,3) (only for style 1)

Definition at line 66 of file vac_vars.f90.

C.60.3.11 ind_p

```
integer, dimension(2) vac_vars::vac_type::ind_p
```

index of local process in grid

Definition at line 57 of file vac_vars.f90.

C.60.3.12 jq

```
real(dp) vac_vars::vac_type::jq
```

iota (tor. flux) or q (pol. flux) at edge

Definition at line 62 of file vac_vars.f90.

C.60.3.13 lims_c

```
integer, dimension(:,:), allocatable vac_vars::vac_type::lims_c
```

column limits for different subrows of G and H

Definition at line 60 of file vac_vars.f90.

C.60.3.14 lims_r

```
integer, dimension(:,:), allocatable vac_vars::vac_type::lims_r
```

row limits for different subcolumns of G and H

Definition at line 61 of file vac_vars.f90.

C.60.3.15 mpi_comm

```
integer vac_vars::vac_type::mpi_comm
```

communicator for vacuum

Definition at line 52 of file vac_vars.f90.

C.60.3.16 n_ang

```
integer, dimension(2) vac_vars::vac_type::n_ang
```

number of angles (1) and number of field lines (2)

Definition at line 56 of file vac_vars.f90.

C.60.3.17 n_bnd

```
integer vac_vars::vac_type::n_bnd
```

number of points in boundary

Definition at line 50 of file vac_vars.f90.

C.60.3.18 n_loc

```
integer, dimension(2) vac_vars::vac_type::n_loc
```

local number of rows and columns

Definition at line 58 of file vac_vars.f90.

C.60.3.19 n_p

```
integer, dimension(2) vac_vars::vac_type::n_p
```

nr. of processes in grid

Definition at line 55 of file vac_vars.f90.

C.60.3.20 norm

```
real(dp), dimension(:,:), allocatable vac_vars::vac_type::norm
```

J nabla psi normal vector.

Definition at line 64 of file vac_vars.f90.

C.60.3.21 prim_x

```
integer vac_vars::vac_type::prim_x
```

primary mode number

Definition at line 48 of file vac_vars.f90.

C.60.3.22 res

```
complex(dp), dimension(:,:), allocatable vac_vars::vac_type::res
```

vacuum response

Definition at line 70 of file vac_vars.f90.

C.60.3.23 sec_x

```
integer, dimension(:), allocatable vac_vars::vac_type::sec_x
```

secondary mode numbers

Definition at line 59 of file vac_vars.f90.

C.60.3.24 style

```
integer vac_vars::vac_type::style
```

style of vacuum (1: field-line 3-D, 2: axisymmetric)

Definition at line 47 of file vac_vars.f90.

C.60.3.25 x_vec

```
real(dp), dimension(:,:), allocatable vac_vars::vac_type::x_vec
```

Cartesian vector of position.

Definition at line 67 of file vac_vars.f90.

C.61 hdf5_vars::var_1d_type Type Reference

1D equivalent of multidimensional variables, used for internal HDF5 storage.

Public Attributes

- real(dp), dimension(:), allocatable p
  1D equivalent of data of variable
- integer, dimension(:), allocatable tot_i_min
  total min.of indices of variable
- integer, dimension(:), allocatable tot_i_max
  total max.of indices of variable
- integer, dimension(:), allocatable loc_i_min
  group min.of indices of variable
- integer, dimension(:), allocatable loc_i_max
  group max.of indices of variable
- character(len=max_str_ln) var_name
  name of variable

C.61.1 Detailed Description

1D equivalent of multidimensional variables, used for internal HDF5 storage.

Definition at line 48 of file HDF5_vars.f90.

C.61.2 Member Data Documentation

C.61.2.1 loc_i_max

```
integer, dimension(:), allocatable hdf5_vars::var_1d_type::loc_i_max
```

group max.of indices of variable

Definition at line 53 of file HDF5_vars.f90.

C.61.2.2 loc_i_min

```
integer, dimension(:), allocatable hdf5_vars::var_1d_type::loc_i_min
```

group min.of indices of variable

Definition at line 52 of file HDF5_vars.f90.

C.61.2.3 p

```
real(dp), dimension(:), allocatable hdf5_vars::var_1d_type::p
```

1D equivalent of data of variable

Definition at line 49 of file HDF5_vars.f90.

C.61.2.4 tot_i_max

```
integer, dimension(:), allocatable hdf5_vars::var_1d_type::tot_i_max
```

total max.of indices of variable

Definition at line 51 of file HDF5_vars.f90.

C.61.2.5 tot_i_min

```
integer, dimension(:), allocatable hdf5_vars::var_1d_type::tot_i_min
```

total min.of indices of variable

Definition at line 50 of file HDF5_vars.f90.

C.61.2.6 var_name

```
character(len=max_str_ln) hdf5_vars::var_1d_type::var_name
```

name of variable

Definition at line 54 of file HDF5_vars.f90.

C.62 x_vars::x_1_type Type Reference

vectorial perturbation type

Public Member Functions

- procedure init (x, mds, grid_x, lim_sec_x)
  initialize
- procedure copy (x_i, mds, grid_i, x_o)
  copy
- procedure dealloc (x)
  deallocate

Public Attributes

- integer n_mod
  size of n and m (nr. of modes)
- integer, dimension(2) lim_sec_x
  limits of m_X (pol. flux) or n_X (tor. flux)
- integer, dimension(:,:), allocatable n
  vector of poloidal mode numbers
- integer, dimension(:,:), allocatable m
  vector of poloidal mode numbers
- complex(dp), dimension(:,:,:,:), allocatable u_0
  U_m^0
- complex(dp), dimension(:,:,:,:), allocatable u_1
  U_m^1
- complex(dp), dimension(:,:,:,:), allocatable du_0
  $\mathcal{J}\vec{B}\cdot\nabla U_m^0$
- complex(dp), dimension(:,:,:,:), allocatable du_1
  $\mathcal{J}\vec{B}\cdot\nabla U_m^1$
- real(dp) estim_mem_usage
  estimated memory usage

C.62.1 Detailed Description

vectorial perturbation type

The arrays here are of the form:

- `U_x_i` and DU_X_i: `(1:angle_1,1:angle_2,1;n_mod)`

See also

See grid_vars.grid_type for a discussion on `ang_1` and `ang_2`.

Definition at line 51 of file X_vars.f90.

C.62.2 Member Function/Subroutine Documentation

C.62.2.1 copy()

```
procedure x_vars::x_1_type::copy (
            class(x_1_type), intent(in) x_i,
            type(modes_type), intent(in) mds,
            type(grid_type), intent(in) grid_i,
            type(x_1_type), intent(inout) x_o)
```

copy

Parameters

in	*x_i*	X_1 to be copied
in	*mds*	general modes variables
in	*grid↩_i*	grid of eq_i
in,out	*x_o*	copied X_1

Definition at line 67 of file X_vars.f90.

C.62.2.2 dealloc()

```
procedure x_vars::x_1_type::dealloc (
            class(x_1_type), intent(inout) x)
```

deallocate

Parameters

in,out	*x*	perturbation variables to be deallocated

Definition at line 69 of file X_vars.f90.

C.62.2.3 init()

```
procedure x_vars::x_1_type::init (
                class(x_1_type), intent(inout) x,
                type(modes_type), intent(in) mds,
                type(grid_type), intent(in) grid_x,
                integer, dimension(2), intent(in), optional lim_sec_x)
```

initialize

Allocates the variables, the number of modes, as well as `n` and `m`.

Optionally, the secondary mode numbers can be specified (`m` if poloidal flux is used and `n` if toroidal flux). By default, they are taken from the global `X_vars` variables.

Note

If the lowest limits of the grid is not 1 (e.g. `grid_soli_min = 1` for first process), the input variable `i_min` should be set to set correctly. For a full grid, it should be set to 1.

Parameters

`in,out`	*x*	vectorial perturbation variables
`in`	*mds*	general modes variables
`in`	*grid_x*	perturbation grid
`in`	*lim_sec_x*	limits of `m_X` (pol. flux) or `n_X` (tor. flux)

Definition at line 65 of file X_vars.f90.

C.62.3 Member Data Documentation

C.62.3.1 du_0

```
complex(dp), dimension(:,:,:,:), allocatable x_vars::x_1_type::du_0
```

$\mathcal{J}\vec{B}\cdot\nabla U_m^0$

Definition at line 58 of file X_vars.f90.

C.62.3.2 du_1

```
complex(dp), dimension(:,:,:,:), allocatable x_vars::x_1_type::du_1
```

$\mathcal{J}\vec{B}\cdot\nabla U_m^1$

Definition at line 59 of file X_vars.f90.

C.62.3.3 estim_mem_usage

```
real(dp) x_vars::x_1_type::estim_mem_usage
```

estimated memory usage

Note

Debug version only

Definition at line 61 of file X_vars.f90.

C.62.3.4 lim_sec_x

```
integer, dimension(2) x_vars::x_1_type::lim_sec_x
```

limits of `m_X` (pol. flux) or `n_X` (tor. flux)

Definition at line 53 of file X_vars.f90.

C.62.3.5 m

```
integer, dimension(:,:), allocatable x_vars::x_1_type::m
```

vector of poloidal mode numbers

Definition at line 55 of file X_vars.f90.

C.62.3.6 n

```
integer, dimension(:,:), allocatable x_vars::x_1_type::n
```

vector of poloidal mode numbers

Definition at line 54 of file X_vars.f90.

C.62.3.7 n_mod

```
integer x_vars::x_1_type::n_mod
```

size of n and m (nr. of modes)

Definition at line 52 of file X_vars.f90.

C.62.3.8 u_0

```
complex(dp), dimension(:,:,:,:), allocatable x_vars::x_1_type::u_0
```

U_m^0

Definition at line 56 of file X_vars.f90.

C.62.3.9 u_1

```
complex(dp), dimension(:,:,:,:), allocatable x_vars::x_1_type::u_1
```

U_m^1

Definition at line 57 of file X_vars.f90.

C.63 x_vars::x_2_type Type Reference

tensorial perturbation type

Public Member Functions

- procedure init (x, mds, grid_x, lim_sec_x, is_field_averaged)
  initialize
- procedure copy (x_i, mds, grid_i, x_o)
  copy
- procedure dealloc (x)
  deallocate

Public Attributes

- integer, dimension(2) n_mod
  size of n and m (nr. of modes)
- integer, dimension(2, 2) lim_sec_x
  limits of m_X (pol. flux) or n_X (tor. flux)
- integer, dimension(:,:), allocatable n_1
  vector of toroidal mode numbers of dimension 1
- integer, dimension(:,:), allocatable n_2
  vector of toroidal mode numbers of dimension 2
- integer, dimension(:,:), allocatable m_1
  vector of poloidal mode numbers of dimension 1
- integer, dimension(:,:), allocatable m_2
  vector of poloidal mode numbers of dimension 2
- complex(dp), dimension(:,:,:,:), allocatable pv_0
  $\widetilde{PV}^0$ *coefficient*

- complex(dp), dimension(:,:,:,:), allocatable pv_1
  $\widetilde{PV}^1$ *coefficient*
- complex(dp), dimension(:,:,:,:), allocatable pv_2
  $\widetilde{PV}^2$ *coefficient*
- complex(dp), dimension(:,:,:,:), allocatable kv_0
  $\widetilde{KV}^0$ *coefficient*
- complex(dp), dimension(:,:,:,:), allocatable kv_1
  $\widetilde{KV}^1$ *coefficient*
- complex(dp), dimension(:,:,:,:), allocatable kv_2
  $\widetilde{KV}^2$ *coefficient*
- real(dp) estim_mem_usage
  estimated memory usage

C.63.1 Detailed Description

tensorial perturbation type

The arrays here are of the form:

- `PV_i` and KV_i: `(1:angle_1,1:angle_2,1;n_mod^2)`

See also

See grid_vars.grid_type for a discussion on `ang_1` and `ang_2`.

Note

This type is also used for field-averaged tensorial perturbation variables, with `angle_1` of size 1.

Definition at line 81 of file X_vars.f90.

C.63.2 Member Function/Subroutine Documentation

C.63.2.1 copy()

```
procedure x_vars::x_2_type::copy (
            class(x_2_type), intent(in) x_i,
            type(modes_type), intent(in) mds,
            type(grid_type), intent(in) grid_i,
            type(x_2_type), intent(inout) x_o)
```

copy

Parameters

in	*x_i*	X_2 to be copied

in	*mds*	general modes variables
in	*grid↩ _i*	grid of eq_i
in,out	*x_o*	copied X_1

Definition at line 101 of file X_vars.f90.

C.63.2.2 dealloc()

```
procedure x_vars::x_2_type::dealloc (
                class(x_2_type), intent(inout) x)
```

deallocate

Parameters

in,out	*x*	perturbation variables to be deallocated

Definition at line 103 of file X_vars.f90.

C.63.2.3 init()

```
procedure x_vars::x_2_type::init (
                class(x_2_type), intent(inout) x,
                type(modes_type), intent(in) mds,
                type(grid_type), intent(in) grid_x,
                integer, dimension(2,2), intent(in), optional lim_sec_x,
                logical, intent(in), optional is_field_averaged)
```

initialize

See also

See init_X_1().

Note

The tensorial perturbation type can also be used for field-aligned variables, in which case the first index is assumed to have dimension 1 only. This can be triggered using `is_field_averaged`. There is no difference between a tensorial perturbation type with size of first dimension set to one through the use of this flag, or through other means.

Parameters

in,out	*x*	tensorial perturbation variables

in	*mds*	general modes variables
in	*grid_x*	perturbation grid
in	*lim_sec_x*	limits of m_X (pol. flux) or n_X (tor. flux) for both dimensions
in	*is_field_averaged*	if field-aligned, only one dimension for first index

Definition at line 99 of file X_vars.f90.

C.63.3 Member Data Documentation

C.63.3.1 estim_mem_usage

```
real(dp) x_vars::x_2_type::estim_mem_usage
```

estimated memory usage

Note

Debug version only

Definition at line 95 of file X_vars.f90.

C.63.3.2 kv_0

```
complex(dp), dimension(:,:,:,:), allocatable x_vars::x_2_type::kv_0
```

$\widetilde{KV}^0$ coefficient

Definition at line 91 of file X_vars.f90.

C.63.3.3 kv_1

```
complex(dp), dimension(:,:,:,:), allocatable x_vars::x_2_type::kv_1
```

$\widetilde{KV}^1$ coefficient

Definition at line 92 of file X_vars.f90.

C.63.3.4 kv_2

```
complex(dp), dimension(:,:,:,:), allocatable x_vars::x_2_type::kv_2
```

$\widetilde{KV}^2$ coefficient

Definition at line 93 of file X_vars.f90.

C.63.3.5 lim_sec_x

```
integer, dimension(2,2) x_vars::x_2_type::lim_sec_x
```

limits of `m_X` (pol. flux) or `n_X` (tor. flux)

Definition at line 83 of file X_vars.f90.

C.63.3.6 m_1

```
integer, dimension(:,:), allocatable x_vars::x_2_type::m_1
```

vector of poloidal mode numbers of dimension 1

Definition at line 86 of file X_vars.f90.

C.63.3.7 m_2

```
integer, dimension(:,:), allocatable x_vars::x_2_type::m_2
```

vector of poloidal mode numbers of dimension 2

Definition at line 87 of file X_vars.f90.

C.63.3.8 n_1

```
integer, dimension(:,:), allocatable x_vars::x_2_type::n_1
```

vector of toroidal mode numbers of dimension 1

Definition at line 84 of file X_vars.f90.

C.63.3.9 n_2

```
integer, dimension(:,:), allocatable x_vars::x_2_type::n_2
```

vector of toroidal mode numbers of dimension 2

Definition at line 85 of file X_vars.f90.

C.63.3.10 n_mod

```
integer, dimension(2) x_vars::x_2_type::n_mod
```

size of n and m (nr. of modes)

Definition at line 82 of file X_vars.f90.

C.63.3.11 pv_0

```
complex(dp), dimension(:,:,:,:), allocatable x_vars::x_2_type::pv_0
```

$\widetilde{PV}^0$ coefficient

Definition at line 88 of file X_vars.f90.

C.63.3.12 pv_1

```
complex(dp), dimension(:,:,:,:), allocatable x_vars::x_2_type::pv_1
```

$\widetilde{PV}^1$ coefficient

Definition at line 89 of file X_vars.f90.

C.63.3.13 pv_2

```
complex(dp), dimension(:,:,:,:), allocatable x_vars::x_2_type::pv_2
```

$\widetilde{PV}^2$ coefficient

Definition at line 90 of file X_vars.f90.

C.64 hdf5_vars::xml_str_type Type Reference

XML strings used in XDMF.

Public Attributes

- character(len=max_str_ln) name
  name of this item
- integer max_xml_ln = 300
  max. length of xml string
- character(len=max_xml_ln), dimension(:), allocatable xml_str
  XML string.

C.64.1 Detailed Description

XML strings used in XDMF.

Definition at line 33 of file HDF5_vars.f90.

C.64.2 Member Data Documentation

C.64.2.1 max_xml_ln

```
integer hdf5_vars::xml_str_type::max_xml_ln = 300
```

max. length of xml string

Definition at line 35 of file HDF5_vars.f90.

C.64.2.2 name

```
character(len=max_str_ln) hdf5_vars::xml_str_type::name
```

name of this item

Definition at line 34 of file HDF5_vars.f90.

C.64.2.3 xml_str

```
character(len=max_xml_ln), dimension(:), allocatable hdf5_vars::xml_str_type::xml_str
```

XML string.

Definition at line 36 of file HDF5_vars.f90.

Appendix D

File Documentation

D.1 Documentation/faq.dox File Reference

D.2 Documentation/inputs.dox File Reference

D.3 Documentation/installation.dox File Reference

D.4 Documentation/mainpage.dox File Reference

Variables

- The last line will be deleted in html

D.4.1 Variable Documentation

D.4.1.1 html

```
The last line will be deleted in html
```

Definition at line 35 of file mainpage.dox.

D.5 Documentation/outputs.dox File Reference

D.6 Documentation/overview.dox File Reference

D.7 Documentation/tutorial.dox File Reference

D.8 /Users/toonweyens/Code/PB3D/Modules/driver_eq.f90 File Reference

Modules

- module driver_eq

  Driver of the equilibrium part of PB3D.

Functions/Subroutines

- integer function, public driver_eq::run_driver_eq (grid_eq_out, grid_eq_b_out, eq_1_out, eq_2_out, vac)

  Main driver of PB3D equilibrium part.

D.9 driver_eq.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Driver of the equilibrium part of PB3D.
!-----------------------------------------------------------------------------!
module driver_eq
#include <PB3D_macros.h>
    use str_utilities
    use output_ops
    use messages
    use num_vars, only: dp, pi, max_str_ln
    use grid_vars, only: grid_type
    use eq_vars, only: eq_1_type, eq_2_type
    use vac_vars, only: vac_type

    implicit none
    private
    public run_driver_eq

    ! global variables
#if ldebug
    logical :: plot_info = .false.                                      !< plot information for comparison with HELENA \ldebug
#endif

contains
    !> Main driver of PB3D equilibrium part.
    !!
    !!  - sets up ([out] means for output):
    !!      * \c grid_eq [out] (for HELENA, only first Richardson level)
    !!      * \c grid_eq_B [out] (for VMEC, equal to grid_eq_out)
    !!      * \c eq_1 [out] (only first Richardson level)
    !!      * \c eq_2 [out] (for HELENA, only first Richardson level)
    !!  where output means
    !!      * on the equilibrium grid if \c X_grid style is 1,3 (no change).
    !!      * on a redistributed grid if \c X_grid_style is 2
    !!
```

```
    !!  - writes to HDF5:
    !!      * \c grid_eq (for HELENA, only first Richardson level)
    !!      * \c grid_eq_B (for VMEC, equal to grid_eq)
    !!      * \c eq_1 (only first Richardson level)
    !!      * \c eq_2 (only for HELENA)
    !!
    !!  - deallocates:
    !!      * \c grid_eq [out] before setting up
    !!      * \c grid_B_eq [out] before setting up
    !!      * \c eq_2 [out] before setting up
    !!
    !! \return ierr
    integer function run_driver_eq(grid_eq_out,grid_eq_B_out,eq_1_out,&
        &eq_2_out,vac) result(ierr)

        use num_vars, only: use_pol_flux_f, eq_style, plot_flux_q, &
            &plot_magn_grid, plot_b, plot_j, plot_kappa, eq_job_nr, &
            &eq_jobs_lims, jump_to_sol, rich_restart_lvl, ltest, alpha_style, &
            &x_grid_style
        use eq_ops, only: calc_eq, print_output_eq, flux_q_plot, &
            &redistribute_output_eq, b_plot, j_plot, kappa_plot
        use grid_ops, only: setup_grid_eq_b, print_output_grid, &
            &calc_norm_range, setup_grid_eq, calc_ang_grid_eq_b, &
            &magn_grid_plot, redistribute_output_grid
        use pb3d_ops, only: reconstruct_pb3d_grid, reconstruct_pb3d_eq_1, &
            &reconstruct_pb3d_eq_2
        use grid_vars, only: min_par_x, max_par_x, n_alpha, alpha
        use num_utilities, only: derivs
        use mpi_utilities, only: wait_mpi
        use rich_vars, only: rich_lvl
        use hdf5_ops, only: create_output_hdf5
        use vac_ops, only: store_vac
        !!use num_utilities, only: calc_aux_utilities

        character(*), parameter :: rout_name = 'run_driver_eq'

        ! input / output
        type(grid_type), intent(inout), target :: grid_eq_out                   !< output equilibrium grid
        type(grid_type), intent(inout), pointer :: grid_eq_b_out                !< output field-aligned equilibrium grid
        type(eq_1_type), intent(inout) :: eq_1_out                              !< flux equilibrium variables in output grid
        type(eq_2_type), intent(inout) :: eq_2_out                              !< metric equilibrium variables in output grid
        type(vac_type), intent(inout) :: vac                                    !< vacuum variables

        ! local variables
        type(eq_1_type) :: eq_1                                                 ! flux equilibrium variables
        type(eq_2_type) :: eq_2                                                 ! metric equilibrium variables
        type(grid_type) :: grid_eq                                              ! equilibrium grid
        type(grid_type) :: grid_eq_b                                            ! field-aligned equilibrium grid
        integer :: eq_limits(2)                                                 ! min. and max. index of eq. grid of this process
        integer :: rich_lvl_name                                                ! either the Richardson level or zero, to append to names
        logical :: do_eq_1_ops                                                  ! whether specific calculations for eq_1 are necessary
        logical :: do_eq_2_ops                                                  ! whether specific calculations for eq_2 are necessary
        logical :: only_half_grid                                               ! calculate only half grid
        logical :: dealloc_vars                                                 ! whether to deallocate variables to save memory
        character(len=8) :: flux_name                                           ! toroidal or poloidal
        character(len=max_str_ln) :: grid_eq_b_name                             ! name of grid_eq_B

        ! initialize ierr
        ierr = 0

        ! decide whether to append Richardson level to name
        select case (eq_style)
            case (1)                                                            ! VMEC
                rich_lvl_name = rich_lvl                                        ! append richardson level
            case (2)                                                            ! HELENA
                rich_lvl_name = 0                                               ! do not append name
        end select

        ! Divide equilibrium grid under  group processes, calculating the limits
        ! and the normal coordinate.
        ierr = calc_norm_range('PB3D_eq',eq_limits=eq_limits)
        chckerr('')

        ! jump to solution if requested
        if (rich_lvl.eq.rich_restart_lvl .and. jump_to_sol) then
            call writo('Prepare to jump to solution')
            call lvl_ud(1)

            ! grid_eq_out
            select case (x_grid_style)
                case (1,3)                                                      ! equilibrium or optimized
```

```
                    ! perturbation  quantities  are  calculated  on  equilibrium
                    ! normal grid or optimized variant
                    ierr = reconstruct_pb3d_grid(grid_eq_out,'eq',&
                        &rich_lvl=rich_lvl_name,grid_limits=eq_limits)
                    chckerr('')
                case (2)                                                    ! solution
                    ! start from equilibrium grid in equilibrium limits and then
                    ! redistribute it
                    ierr = reconstruct_pb3d_grid(grid_eq,'eq',&
                        &rich_lvl=rich_lvl_name,grid_limits=eq_limits)
                    chckerr('')
                    ierr = redistribute_output_grid(grid_eq,grid_eq_out)
                    chckerr('')
                    call grid_eq%dealloc()
            end select

            ! eq_1_out
            ierr = reconstruct_pb3d_eq_1(grid_eq_out,eq_1_out,'eq_1')
            chckerr('')

            ! eq_2_out
            if (eq_style.eq.2) then
                ierr = reconstruct_pb3d_eq_2(grid_eq_out,eq_2_out,'eq_2')
                chckerr('')
            end if

            call lvl_ud(-1)
            call writo('Skipping rest to jump to solution')
            return
        end if

        ! set up whether to deallocate variables
        dealloc_vars = .true.
#if ldebug
        if (plot_info) dealloc_vars = .false.
        if (ltest) dealloc_vars = .false.
#endif
        if (plot_b .or. plot_j .or. plot_kappa) dealloc_vars = .false.          ! need transformation matrices

        ! user output
        call writo('The equilibrium variables are processed')
        call lvl_ud(1)

        select case (alpha_style)
            case (1)                                                        ! single field line, multiple turns
                call writo('With a single field-line')
                call lvl_ud(1)
                call writo('with label alpha = '//&
                    &trim(r2strt(alpha(1))))
                call writo('with parallel range '//trim(r2strt(min_par_x))//&
                    &'..'//trim(r2strt(max_par_x)))
                call lvl_ud(-1)
            case (2)                                                        ! multiple field lines, single turns
                call writo('With '//trim(i2str(n_alpha))//' field-lines')
        end select
        if (use_pol_flux_f) then
            flux_name = 'poloidal'
        else
            flux_name = 'toroidal'
        end if
        call writo('using the '//trim(flux_name)//' flux as the normal &
            &variable')

        call lvl_ud(-1)

        ! set up whether half or full parallel grid has to be calculated
        if (rich_lvl.eq.1) then
            only_half_grid = .false.
        else
            only_half_grid = .true.
        end if

        ! decide whether to do certain equilibrium calculations
        if (rich_lvl.eq.rich_restart_lvl .and. eq_job_nr.eq.1) then
            do_eq_1_ops = .true.
        else
            do_eq_1_ops = .false.
        end if
        select case (eq_style)
            case (1)                                                        ! VMEC
                do_eq_2_ops = .true.
            case (2)                                                        ! HELENA
                if (rich_lvl.eq.rich_restart_lvl .and. eq_job_nr.eq.1) then
                    do_eq_2_ops = .true.
                else
                    do_eq_2_ops = .false.
                end if
```

```
        end select

        ! setup equilibrium grid
        call writo('Determine the equilibrium grid')
        call lvl_ud(1)
        ierr = setup_grid_eq(grid_eq,eq_limits)
        call lvl_ud(-1)
        chckerr('')

        ! Calculate the flux equilibrium quantities.
        ! Note: There  are some E  quantities that are needed  in to set  up the
        ! equilibrium grid and that are not present in eq_1_out.
        ierr = calc_eq(grid_eq,eq_1)
        chckerr('')

        ! Write to HDF5 only for first calculation
        if (rich_lvl.eq.1 .and. eq_job_nr.eq.1) then
            ! write flux equilibrium variables to output
            ierr = print_output_eq(grid_eq,eq_1,'eq_1')
            chckerr('')

            ! plot flux quantities if requested
            if (plot_flux_q) then
                ierr = flux_q_plot(grid_eq,eq_1)
                chckerr('')
            else
                call writo('Flux quantities plot not requested')
            end if
        end if

        ! grid_eq and grid_eq_B
        ! VMEC: The equilibrium grid is  field-aligned but its angular variables
        ! need to be set up still.
        ! HELENA:  The equilibrium  grid  is not  field-aligned  but taken  from
        ! the  equilibrium output.  The  field-aligned  grid has  to  be set  up
        ! separately.
        select case (eq_style)
            case (1)                                                        ! VMEC
                ! calculate angular grid points for equilibrium grid
                call writo('Calculate angular equilibrium grid')
                ierr = calc_ang_grid_eq_b(grid_eq,eq_1,only_half_grid=&
                    &only_half_grid)
                chckerr('')
            case (2)                                                        ! HELENA
                ! set up field-aligned equilibrium grid
                call writo('Determine the field-aligned equilibrium grid')
                ierr = setup_grid_eq_b(grid_eq,grid_eq_b,eq_1,&
                    &only_half_grid=only_half_grid)
                chckerr('')

                ! write field-aligned equilibrium grid variables to output
                ierr = print_output_grid(grid_eq_b,'field-aligned equilibrium',&
                    &'eq_B',rich_lvl=rich_lvl,par_div=.true.)
                chckerr('')
        end select

        ! only do when needed
        if (do_eq_2_ops) then
            ! write equilibrium grid variables to output
            ierr = print_output_grid(grid_eq,'equilibrium','eq',&
                &rich_lvl=rich_lvl_name,par_div=size(eq_jobs_lims,2).gt.1)
            chckerr('')

            ! Calculate the metric equilibrium quantities
            ierr = calc_eq(grid_eq,eq_1,eq_2,dealloc_vars=dealloc_vars)
            chckerr('')

#if ldebug
            if (plot_info) then
                ! plot info for comparison between VMEC and HELENA
                call plot_info_for_vmec_hel_comparision()
            end if
#endif

            if (rich_lvl.eq.1 .and. eq_style.eq.2) then                     ! only if first Richardson level
                ! write metric equilibrium variables to output
                ierr = print_output_eq(grid_eq,eq_2,'eq_2',par_div=.false.)
                chckerr('')
            end if

            ! store vacuum variables
            ierr = store_vac(grid_eq,eq_1,eq_2,vac)
            chckerr('')
        end if

        ! set output variables
        select case (x_grid_style)
```

```
            case (1,3)                                                  ! equilibrium or enriched
                call writo('Copy the equilibrium grids and variables to &
                    &output grid')
            case (2)                                                    ! solution
                call writo('Redistribute the equilibrium grids and variables &
                    &to output grid')
        end select
        call lvl_ud(1)

        ! grid_eq_out
        if (do_eq_2_ops) then
            if (associated(grid_eq_out%r_F)) call grid_eq_out%dealloc()         ! deallocate if necessary
            select case (x_grid_style)
                case (1,3)                                              ! equilibrium or enriched
                    ierr = grid_eq%copy(grid_eq_out)
                    chckerr('')
                case (2)                                                ! solution
                    ierr = redistribute_output_grid(grid_eq,grid_eq_out)
                    chckerr('')
            end select
        end if

        ! eq_1_out
        if (do_eq_1_ops) then
            if (allocated(eq_1_out%pres_FD)) call eq_1_out%dealloc()            ! deallocate if necessary
            select case (x_grid_style)
                case (1,3)                                              ! equilibrium or optimized
                    call eq_1%copy(grid_eq,eq_1_out)
                case (2)                                                ! solution
                    ierr = redistribute_output_eq(grid_eq,grid_eq_out,eq_1,&
                        &eq_1_out)
                    chckerr('')
            end select
        end if

        ! eq_2_out
        if (do_eq_2_ops) then
            if (allocated(eq_2_out%jac_FD)) call eq_2_out%dealloc()             ! deallocate if necessary
            select case (x_grid_style)
                case (1,3)                                              ! equilibrium or optimized
                    call eq_2%copy(grid_eq,eq_2_out)
                case (2)                                                ! solution
                    ierr = redistribute_output_eq(grid_eq,grid_eq_out,eq_2,&
                        &eq_2_out)
                    chckerr('')
            end select
        end if

        ! grid_eq_B_out
        select case (eq_style)
            case (1)                                                    ! VMEC
                grid_eq_b_out => grid_eq_out
            case (2)                                                    ! HELENA
                if (associated(grid_eq_b_out)) then
                    call grid_eq_b_out%dealloc()
                    deallocate(grid_eq_b_out)
                    nullify(grid_eq_b_out)
                end if
                allocate(grid_eq_b_out)
                select case (x_grid_style)
                    case (1,3)                                          ! equilibrium or optimized
                        ierr = grid_eq_b%copy(grid_eq_b_out)
                        chckerr('')
                    case (2)                                            ! solution
                        ierr = redistribute_output_grid(grid_eq_b,grid_eq_b_out)
                        chckerr('')
                end select
        end select

        ! clean up
        call grid_eq%dealloc()
        call eq_1%dealloc()                                             ! eq_1 is always calculated because there
    are some quantities that are not present in eq_1_out
        if (do_eq_2_ops) then
            call eq_2%dealloc()
        end if
        if (eq_style.eq.2) then
            call grid_eq_b%dealloc()
        end if

        call lvl_ud(-1)

        ! plot magnetic field if requested
        ! (done in parts, for every parallel job)
        if (plot_b) then
            select case (eq_style)
                case (1)                                                ! VMEC
```

```
                    ierr = b_plot(grid_eq_out,eq_1_out,eq_2_out,&
                        &rich_lvl=rich_lvl)
                    chckerr('')
                case (2)                                                ! HELENA
                    if (rich_lvl.eq.1) then
                        ierr = b_plot(grid_eq_out,eq_1_out,eq_2_out)
                        chckerr('')
                    end if
            end select
        else
            if (eq_job_nr.eq.1) call writo('Magnetic field plot not requested')
        end if

        ! plot current if requested
        ! (done in parts, for every parallel job)
        if (plot_j) then
            select case (eq_style)
                case (1)                                                ! VMEC
                    ierr = j_plot(grid_eq_out,eq_1_out,eq_2_out,&
                        &rich_lvl=rich_lvl)
                    chckerr('')
                case (2)                                                ! HELENA
                    if (rich_lvl.eq.1) then
                        ierr = j_plot(grid_eq_out,eq_1_out,eq_2_out)
                        chckerr('')
                    end if
            end select
        else
            if (eq_job_nr.eq.1) call writo('Current plot not requested')
        end if

        ! plot curvature if requested
        ! (done in parts, for every parallel job)
        if (plot_kappa) then
            select case (eq_style)
                case (1)                                                ! VMEC
                    ierr = kappa_plot(grid_eq_out,eq_1_out,eq_2_out,&
                        &rich_lvl=rich_lvl)
                    chckerr('')
                case (2)                                                ! HELENA
                    if (rich_lvl.eq.1) then
                        ierr = kappa_plot(grid_eq_out,eq_1_out,eq_2_out)
                        chckerr('')
                    end if
            end select
        else
            if (eq_job_nr.eq.1) call writo('Curvature plot not requested')
        end if

        ! plot full field-aligned grid if requested
        ! Note: As  this needs a total  grid, it cannot be  easiliy created like
        ! plot_B, where the output is updated for every equilibrium job.
        if (plot_magn_grid) then
            if (eq_job_nr.eq.size(eq_jobs_lims,2)) then
                ! set up the name of grid_eq_B
                select case (eq_style)
                    case (1)                                            ! VMEC
                        grid_eq_b_name = 'eq'                           ! already field-aligned
                    case (2)                                            ! HELENA
                        grid_eq_b_name = 'eq_B'                         ! not already field-aligned
                end select

                ! reconstruct the full field-aligned grid
                ierr = reconstruct_pb3d_grid(grid_eq_b,trim(grid_eq_b_name),&
                    &rich_lvl=rich_lvl,tot_rich=.true.,grid_limits=eq_limits)
                chckerr('')

                ! wait for all processes (sometimes necessary to finish lock)
                ierr = wait_mpi()
                chckerr('')

                ! plot it
                ierr = magn_grid_plot(grid_eq_b)
                chckerr('')

                ! deallocate
                call grid_eq_b%dealloc()
            end if
        else
            if (eq_job_nr.eq.1) call writo('Magnetic grid plot not requested')
        end if
#if ldebug
    contains
        ! Plots information for comparison between HELENA and VMEC.
        !> \private
        subroutine plot_info_for_vmec_hel_comparision()
            use helena_vars, only: r_h, z_h
```

```fortran
        use input_utilities, only: pause_prog, get_int, get_log

        ! local variables
        real(dp), allocatable :: x_plot(:,:,:), y_plot(:,:,:), z_plot(:,:,:)
        integer :: id
        integer :: d(3)
        logical :: not_ready

        call writo('Plotting information for comparison between VMEC and &
            &HELENA',alert=.true.)
        not_ready = .true.
        do while (not_ready)
            call writo('derivative in dim 1?')
            d(1) =  get_int(lim_lo=0)
            call writo('derivative in dim 2?')
            d(2) =  get_int(lim_lo=0)
            call writo('derivative in dim 3?')
            d(3) =  get_int(lim_lo=0)

            ! x, y, z
            allocate(x_plot(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
            allocate(y_plot(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
            allocate(z_plot(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
            x_plot = grid_eq%theta_F
            y_plot = 0._dp
            do id = 1,grid_eq%loc_n_r
                z_plot(:,:,id) = grid_eq%loc_r_F(id)/maxval(grid_eq%r_F)
            end do
            call plot_hdf5('x_plot','x_plot',x_plot)
            call plot_hdf5('y_plot','y_plot',y_plot)
            call plot_hdf5('z_plot','z_plot',z_plot)

            ! flux quantities
            call print_ex_2d('pres_FD','pres_FD',eq_1%pres_FD(:,d(2)),&
                &x=grid_eq%loc_r_F,draw=.false.)
            call draw_ex(['pres_FD'],'pres_FD',1,1,.false.)
            call print_ex_2d('q_saf_FD','q_saf_FD',eq_1%q_saf_FD(:,d(2)),&
                &x=grid_eq%loc_r_F,draw=.false.)
            call draw_ex(['q_saf_FD'],'q_saf_FD',1,1,.false.)
            call print_ex_2d('flux_p_FD','flux_p_FD',&
                &eq_1%flux_p_FD(:,d(2)),x=grid_eq%loc_r_F,draw=.false.)
            call draw_ex(['flux_p_FD'],'flux_p_FD',1,1,.false.)
            call print_ex_2d('flux_t_FD','flux_t_FD',&
                &eq_1%flux_t_FD(:,d(2)),x=grid_eq%loc_r_F,draw=.false.)
            call draw_ex(['flux_t_FD'],'flux_t_FD',1,1,.false.)

            ! R and Z
            select case (eq_style)
                case(1)
                    call plot_hdf5('R_V','R_V',&
                        &eq_2%R_E(:,:,:,d(1),d(2),d(3)),x=x_plot,y=y_plot,&
                        &z=z_plot)
                    call plot_hdf5('Z_V','Z_V',&
                        &eq_2%Z_E(:,:,:,d(1),d(2),d(3)),x=x_plot,y=y_plot,&
                        &z=z_plot)
                    call plot_hdf5('L_V','L_V',&
                        &eq_2%L_E(:,:,:,d(1),d(2),d(3)),x=x_plot,y=y_plot,&
                        &z=z_plot)
                case(2)
                    if (sum(d).eq.0) then
                        call plot_hdf5('R_H','R_H',&
                            &reshape(r_h(:,grid_eq%i_min:grid_eq%i_max),&
                            &[grid_eq%n(1:2),grid_eq%loc_n_r]),&
                            &x=x_plot,y=y_plot,z=z_plot)
                        call plot_hdf5('Z_H','Z_H',&
                            &reshape(z_h(:,grid_eq%i_min:grid_eq%i_max),&
                            &[grid_eq%n(1:2),grid_eq%loc_n_r]),&
                            &x=x_plot,y=y_plot,z=z_plot)
                    else
                        call writo('R and Z can only be plot for d = 0')
                    end if
            end select

            ! metric fators
            call plot_hdf5('jac_FD','jac_FD',&
                &eq_2%jac_FD(:,:,:,d(1),d(2),d(3)),&
                &x=x_plot,y=y_plot,z=z_plot)
            do id = 1,6
                call plot_hdf5('g_FD_'//trim(i2str(id)),&
                    &'g_FD_'//trim(i2str(id)),&
                    &eq_2%g_FD(:,:,:,id,d(1),d(2),d(3)),&
                    &x=x_plot,y=y_plot,z=z_plot)
                call plot_hdf5('h_FD_'//trim(i2str(id)),&
                    &'h_FD_'//trim(i2str(id)),&
                    &eq_2%h_FD(:,:,:,id,d(1),d(2),d(3)),&
                    &x=x_plot,y=y_plot,z=z_plot)
            end do
```

```
                ! clean up
                deallocate(x_plot,y_plot,z_plot)

                call writo('Want to redo the plotting?')
                not_ready = get_log(.true.)
            end do

            ! reset not_ready
            not_ready = .true.

            call writo('Done, paused',alert=.true.)
            call pause_prog()
        end subroutine plot_info_for_vmec_hel_comparision
#endif
    end function run_driver_eq
end module driver_eq
```

D.10 /Users/toonweyens/Code/PB3D/Modules/driver_POST.f90 File Reference

Modules

- module driver_post

  Main driver of PostProcessing of program Peeling Ballooning in 3D.

Functions/Subroutines

- integer function, public driver_post::init_post ()

  Initializes the POST driver.
- integer function, public driver_post::run_driver_post ()

  The main driver routine for postprocessing.
- subroutine, public driver_post::stop_post ()

  Cleans up main driver for postprocessing.

D.11 driver_POST.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Main driver of PostProcessing of program Peeling Ballooning in 3D.
!-----------------------------------------------------------------------------!
module driver_post
#include <PB3D_macros.h>
#include <wrappers.h>
    use str_utilities
    use output_ops
    use messages
    use num_vars, only: max_str_ln, dp, iu, pi
    use grid_vars, only: grid_type
    use eq_vars, only: eq_1_type, eq_2_type
    use x_vars, only: x_1_type, modes_type, &
        &mds_x, mds_sol
    use sol_vars, only: sol_type
    use vac_vars, only: vac_type
    use rich_vars, only: rich_lvl

    implicit none
    private
    public run_driver_post, init_post, stop_post

    ! local variables
```

```
    integer :: lims_norm(2,3)                                          !< normal \c i_limit of eq, X and sol variables
    integer, allocatable :: dum_ser(:)                                 !< serial dummy variable
    integer :: rich_lvl_name                                           !< either the Richardson level or zero, to append to names
    integer :: min_id(3), max_id(3)                                    !< min. and max. index of range 1, 2 and 3
    integer :: last_unstable_id                                        !< index of last unstable EV
    integer :: n_par_tot                                               !< total \c n(1) for all equilibrium jobs together
    character(len=4) :: grid_name(3)                                   !< names of grids
    integer :: n_out(3,2)                                              !< total grid sizes for eq and X grids
    type(sol_type) :: sol                                              !< solution variables
    type(vac_type) :: vac                                              !< vacuum variables
    type(grid_type) :: grid_eq_XYZ                                     !< eq grid for XYZ calculations
    type(grid_type) :: grid_eq_HEL                                     !< HELENA eq grid
    type(grid_type) :: grid_X_HEL                                      !< HELENA X grid
    type(eq_2_type) :: eq_2_HEL                                        !< metric equilibrium for HELENA tables
    type(X_1_type) :: X_HEL                                            !< vectorial perturbation variables for HELENA tables
    complex(dp), allocatable :: E_pot_int(:,:)                         !< \f$\int E_{\text{pot}} \text{d}V\f$ for requested solutions
    complex(dp), allocatable :: E_kin_int(:,:)                         !< \f$\int E_{\text{kin}} \text{d}V\f$ for requested solutions
#if ldebug
    logical :: debug_tor_dep_ripple = .false.                          !< plot debug information for toroidal dependency of ripple \ldebug
#endif

contains
    !> Initializes the POST driver.
    !!
    !!  - set up preliminary variables
    !!      * global variables
    !!      * read grids (full)
    !!      * \c eq_1 (full) and \c n \& \c m (full)
    !!  - set up output grids:
    !!      * <tt> POST_style = 1</tt>: extended grid
    !!      * <tt> POST_style = 2</tt>: field-aligned grid
    !!  - clean up
    !!  - set up final variables
    !!      * normal limits
    !!      *  read grids  (divided), \c  eq_1  (divided) and  \c sol  (divided) variables
    !!  - 1-D output plots
    !!      * resonance plot
    !!      * flux quantities plot
    !!      * magnetic grid plot
    !!  - prepare Eigenvalue plots
    !!      * calculates resonant surfaces
    !!      * plots Eigenvalues
    !!      * finds stability ranges for all Eigenvalues to be plot
    !!      * plots harmonics for every Eigenvalue
    !!      * calculates the parallel ranges of the equilibrium jobs
    !!  - clean up
    !!
    !! In  the actual  driver, more  detailed plots  are possibly  made for  all
    !! requested Eigenvalues if \c full_output.
    !!
    !! \return ierr
    integer function init_post() result(ierr)
        use grid_vars, only: alpha, n_alpha, min_alpha, max_alpha, n_r_sol
        use num_vars, only: post_style, eq_style, rank, plot_magn_grid, &
            &plot_resonance, plot_flux_q, eq_jobs_lims, plot_grid_style, &
            &n_theta_plot, n_zeta_plot, post_output_full, post_output_sol, &
            &compare_tor_pos, min_r_plot, max_r_plot, min_theta_plot, &
            &max_theta_plot, min_zeta_plot, max_zeta_plot, plot_vac_pot, &
            &x_grid_style, n_procs
        use eq_ops, only: flux_q_plot, divide_eq_jobs, calc_eq_jobs_lims
        use pb3d_ops, only: reconstruct_pb3d_in, reconstruct_pb3d_grid, &
            &reconstruct_pb3d_eq_1, reconstruct_pb3d_eq_2, &
            &reconstruct_pb3d_x_1, reconstruct_pb3d_sol, reconstruct_pb3d_vac, &
            &get_pb3d_grid_size
        use vac_ops, only: vac_pot_plot
        use x_ops, only: init_modes, setup_modes, resonance_plot, calc_res_surf
        use grid_ops, only: calc_norm_range, magn_grid_plot
        use grid_utilities, only: calc_eqd_grid
        use helena_vars, only: nchi
        use sol_ops, only: plot_sol_vals, plot_harmonics
        use mpi_utilities, only: wait_mpi, get_ser_var

        character(*), parameter :: rout_name = 'init_POST'

        ! local variables
        type(grid_type), target :: grid_eq                             ! equilibrium grid
        type(grid_type), pointer :: grid_eq_b                          ! field-aligned equilibrium grid
        type(grid_type) :: grid_x                                      ! perturbation grid
        type(grid_type) :: grid_sol                                    ! solution grid
```

```
        type(eq_1_type) :: eq_1                                         ! flux equilibrium
        integer :: n_div                                                ! number of divisions of parallel points
        integer :: id, jd                                               ! counters
        integer :: var_size_without_par(2)                              ! size without parallel dimension for eq_2 and X_1 variables
        integer :: lim_loc(3,2)                                         ! grid ranges for local equilibrium job
        integer :: n_div_max                                            ! maximum divisions of equilibrium jobs
        real(dp), allocatable :: res_surf(:,:)                          ! resonant surfaces
        real(dp), allocatable :: r_f_x(:)                               ! normal points in perturbation grid
        character(len=max_str_ln) :: err_msg                            ! error message

        ! initialize ierr
        ierr = 0

        ! user output
        select case (post_style)
            case (1)                                                    ! extended grid
                call writo('The post-processing will be done in an extended &
                    &grid')
                call lvl_ud(1)
                call writo('range in r: '//trim(r2strt(min_r_plot))//' .. '//&
                    &trim(r2strt(max_r_plot)))
                call writo('range in θ: '//trim(r2strt(min_theta_plot))//&
                    &' .. '//trim(r2strt(max_theta_plot))//', '//&
                    &trim(i2str(n_theta_plot))//' points')
                call writo('range in ζ: '//trim(r2strt(min_zeta_plot))//&
                    &' .. '//trim(r2strt(max_zeta_plot))//', '//&
                    &trim(i2str(n_zeta_plot))//' points')
                call lvl_ud(-1)
            case (2)                                                    ! field-aligned grid
                call writo('The post-processing will be done in the PB3D grid')
                call lvl_ud(1)
                call writo('range in r: '//trim(r2strt(min_r_plot))//' .. '//&
                    &trim(r2strt(max_r_plot)))
                call lvl_ud(-1)
        end select

        ! user output
        call writo('Setting up preliminary variables')
        call lvl_ud(1)

        ! some preliminary things
        ierr = reconstruct_pb3d_in('in')                                ! reconstruct miscellaneous PB3D output variables
        chckerr('')

        ! set up alpha
        allocate(alpha(n_alpha))
        ierr = calc_eqd_grid(alpha,min_alpha*pi,max_alpha*pi,excl_last=.true.)
        chckerr('')

        ! set up whether Richardson level has to be appended to the name
        select case (eq_style)
            case (1)                                                    ! VMEC
                rich_lvl_name = rich_lvl                                ! append richardson level
            case (2)                                                    ! HELENA
                rich_lvl_name = 0                                       ! do not append name
        end select

        ! set up grid names
        select case (eq_style)
            case (1)                                                    ! VMEC
                ! take the standard grids
                grid_name(1) = 'eq'
                grid_name(2) = 'X'
                grid_name(3) = 'sol'
            case (2)                                                    ! HELENA
                ! take the field-aligned grids
                grid_name(1) = 'eq_B'
                grid_name(2) = 'X_B'
                grid_name(3) = 'sol'
        end select

        ! limits to exclude angular part
        lim_loc(1,:) = [0,0]
        lim_loc(2,:) = [0,0]
        lim_loc(3,:) = [1,-1]

        ! reconstruct normal part of full equilibrium, perturbation and solution
        ! grids
        ierr = reconstruct_pb3d_grid(grid_eq,'eq',rich_lvl=rich_lvl_name,&
            &tot_rich=.true.,lim_pos=lim_loc)
        chckerr('')
        ierr = reconstruct_pb3d_grid(grid_x,'X',rich_lvl=rich_lvl_name,&
            &tot_rich=.true.,lim_pos=lim_loc)
        chckerr('')
        ierr = reconstruct_pb3d_grid(grid_sol,'sol')
```

```
        chckerr('')

        ! set up modes variables in full grids
        !  Note: This  is needed  as many  routines count  on this,  for example
        ! 'set_nm_X' in  X_vars, also  used to  initialize a  solution variable,
        ! etc.
        ierr = reconstruct_pb3d_eq_1(grid_eq,eq_1,'eq_1')
        chckerr('')
        ierr = init_modes(grid_eq,eq_1)
        chckerr('')
        ierr = setup_modes(mds_x,grid_eq,grid_x,plot_name='X_POST')
        chckerr('')
        ierr = setup_modes(mds_sol,grid_eq,grid_sol,plot_name='sol_POST')
        chckerr('')

        ! user output
        call lvl_ud(-1)
        call writo('Preliminary variables set up')

        ! user output
        call writo('Dividing grids over MPI processes')
        call lvl_ud(1)

        ! set eq, X and sol limits, using r_F of the grids
        allocate(r_f_x(grid_x%n(3)))                                          ! r_F_X needs to be allocatable for calc_norm_range
        r_f_x = grid_x%r_F
        ierr = calc_norm_range('POST',eq_limits=lims_norm(:,1),&
            &x_limits=lims_norm(:,2),sol_limits=lims_norm(:,3),&
            &r_f_eq=grid_eq%r_F,r_f_x=r_f_x,r_f_sol=grid_sol%r_F)
        chckerr('')
        deallocate(r_f_x)
        call writo('normal grid limits:')
        call lvl_ud(1)
        call writo('proc '//trim(i2str(rank))//' - equilibrium:  '//&
            &trim(i2str(lims_norm(1,1)))//' .. '//trim(i2str(lims_norm(2,1))),&
            &persistent=.true.)
        call writo('proc '//trim(i2str(rank))//' - perturbation: '//&
            &trim(i2str(lims_norm(1,2)))//' .. '//trim(i2str(lims_norm(2,2))),&
            &persistent=.true.)
        call writo('proc '//trim(i2str(rank))//' - solution:     '//&
            &trim(i2str(lims_norm(1,3)))//' .. '//trim(i2str(lims_norm(2,3))),&
            &persistent=.true.)
        call lvl_ud(-1)

        ! synchronize outputs
        ierr = wait_mpi()
        chckerr('')

        ! user output
        call lvl_ud(-1)
        call writo('Grids divided over MPI processes')

        ! clean up
        call grid_eq%dealloc()
        call grid_x%dealloc()
        call grid_sol%dealloc()
        call eq_1%dealloc()

        ! user output
        call writo('Setting up final variables')
        call lvl_ud(1)

        ! reconstruct variables
        ierr = reconstruct_pb3d_grid(grid_eq,'eq',rich_lvl=rich_lvl_name,&
            &tot_rich=.true.,grid_limits=lims_norm(:,1))
        chckerr('')
        ierr = reconstruct_pb3d_grid(grid_x,'X',rich_lvl=rich_lvl_name,&
            &tot_rich=.true.,grid_limits=lims_norm(:,2))
        chckerr('')
        ierr = reconstruct_pb3d_grid(grid_sol,'sol',grid_limits=lims_norm(:,3))
        chckerr('')
        ierr = reconstruct_pb3d_eq_1(grid_eq,eq_1,'eq_1')
        chckerr('')
        if (post_output_sol) then
            ierr = reconstruct_pb3d_sol(mds_sol,grid_sol,sol,'sol',&
                &rich_lvl=rich_lvl)
            chckerr('')
            ierr = 2
            chckerr('Vacuum has not been implemented yet!')
            ierr = reconstruct_pb3d_vac(vac,'vac',rich_lvl=rich_lvl_name)
            chckerr('')
        end if

        ! user output
        call lvl_ud(-1)
        call writo('Final variables set up')
```

```fortran

        ! user output
        call writo('1-D PB3D output plots')
        call lvl_ud(1)

        if (plot_resonance) then
            ierr = resonance_plot(mds_sol,grid_eq,eq_1)
            chckerr('')
        else
            call writo('Resonance plot not requested')
        end if
        if (plot_flux_q) then
            ierr = flux_q_plot(grid_eq,eq_1)
            chckerr('')
        else
            call writo('Flux quantities plot not requested')
        end if
        if (plot_magn_grid) then
            select case (eq_style)
                case (1)                                                    ! VMEC
                    grid_eq_b => grid_eq
                case (2)                                                    ! HELENA
                    allocate(grid_eq_b)
                    ierr = reconstruct_pb3d_grid(grid_eq_b,'eq_B',&
                        &rich_lvl=rich_lvl,tot_rich=.true.,&
                        &grid_limits=lims_norm(:,1))
                    chckerr('')
            end select
            ierr = magn_grid_plot(grid_eq_b)
            chckerr('')
            if (eq_style.eq.2) then
                call grid_eq_b%dealloc()
                deallocate(grid_eq_b)
            end if
            nullify(grid_eq_b)
        else
            call writo('Magnetic grid plot not requested')
        end if

        ierr = wait_mpi()
        chckerr('')

        ! user output
        call lvl_ud(-1)
        call writo('1-D outputs plots done')

        ! user output
        call writo('Prepare 3-D plots')
        call lvl_ud(1)

        ! calculate resonant surfaces on solution grid
        call writo('Calculate resonant surfaces')
        call lvl_ud(1)
        ierr = calc_res_surf(mds_sol,grid_eq,eq_1,res_surf,info=.false.)
        call lvl_ud(-1)

        ! plot solution on solution grid
        if (post_output_sol) then
            call writo('Plot the Eigenvalues')
            call lvl_ud(1)
            call plot_sol_vals(sol,last_unstable_id)
            call lvl_ud(-1)

            ! find stability regions
            call writo('Find stability ranges')
            call lvl_ud(1)
            call find_stab_ranges(sol,min_id,max_id,last_unstable_id)
            call lvl_ud(-1)

            ! plots harmonics for every Eigenvalue, looping over three ranges
            call writo('Plot the Harmonics')
            call lvl_ud(1)
            do jd = 1,3
                if (min_id(jd).le.max_id(jd)) call &
                    &writo('RANGE '//trim(i2str(jd))//': modes '//&
                    &trim(i2str(min_id(jd)))//'..'//trim(i2str(max_id(jd))))
                call lvl_ud(1)

                ! indices in each range
                do id = min_id(jd),max_id(jd)
                    ! user output
                    call writo('Mode '//trim(i2str(id))//'/'//&
                        &trim(i2str(size(sol%val)))//', with eigenvalue '&
                        &//trim(c2strt(sol%val(id))))
                    call lvl_ud(1)
                    ierr = plot_harmonics(mds_sol,grid_sol,sol,id,res_surf)
                    chckerr('')
```

```
                    ! user output
                    call lvl_ud(-1)
                    call writo('Mode '//trim(i2str(id))//'/'//&
                        &trim(i2str(size(sol%val)))//' finished')

                    ! vacuum plots if requested
                    write(*,*) '¡¡¡¡¡ NO VACUUM !!!!!'
                    plot_vac_pot = .false.
                    if (plot_vac_pot) then
                        ! user output
                        call writo('Start vacuum plot for this solution')
                        call lvl_ud(1)

                        ! plot vacuum potential
                        ierr = vac_pot_plot(grid_sol,vac,sol,id)
                        chckerr('')

                        ! user output
                        call lvl_ud(-1)
                        call writo('Done with vacuum plot')
                    end if
                end do

                call lvl_ud(-1)
                if (min_id(jd).le.max_id(jd)) call &
                    &writo('RANGE '//trim(i2str(jd))//' plotted')
            end do
            call lvl_ud(-1)
        end if

        ! Divide  the  equilibrium  jobs (see  subroutine  "calc_memory"  inside
        ! "divide_eq_jobs"), only necessary when full output.
        ! The results are  saved in the global variables  eq_jobs_lims for every
        ! eq_job.
        ! If full output  not requested, allocate eq_jobs_lims to  zero size, so
        ! that the driver is never run.
        if (post_output_full) then
            ! set total grid sizes
            do id = 1,2
                ierr = get_pb3d_grid_size(n_out(:,id),grid_name(id),&
                    &rich_lvl=rich_lvl,tot_rich=.true.)
                chckerr('')
                if (post_style.eq.1) n_out(1:2,id) = [n_theta_plot,n_zeta_plot]
            end do

            ! test if angular grid sizes are compatible
            if (n_out(1,1).ne.n_out(1,2) .or. n_out(2,1).ne.n_out(2,2)) then
                ierr = 1
                call writo('grid sizes for output grids not compatible. This &
                    &is assumed in the calculation of the equilibrium jobs.')
                call writo('If this has changed, adapt and generalize &
                    &"divide_eq_jobs" appropriately.')
                err_msg = 'Take into account how the ranges transfer from &
                    &grid to grid'
                chckerr(err_msg)
            end if

            ! set size of all variables, without parallel dimension
            allocate(dum_ser(2*n_procs))
            ierr = get_ser_var(lims_norm(:,1),dum_ser,scatter=.true.)           ! normal limits of equilibrium grid
            chckerr('')
            var_size_without_par(1) = dum_ser(2*n_procs)-dum_ser(1)+1           ! maximum range
            ierr = get_ser_var(lims_norm(:,2),dum_ser,scatter=.true.)           ! normal limits of perturbation grid
            chckerr('')
            var_size_without_par(2) = dum_ser(2*n_procs)-dum_ser(1)+1           ! maximum range
            select case (x_grid_style)
                case (1,3)                                                      ! equilibrium or enriched
                    var_size_without_par(2) = var_size_without_par(2) + n_r_sol
                case (2)                                                        ! solution
                    ! do nothing
            end select
            deallocate(dum_ser)

            n_div_max = n_out(1,1)-1
            if (compare_tor_pos) n_div_max = 1
            select case (eq_style)
                case (1)                                                        ! VMEC
                    ! divide equilibrium jobs
                    ierr = divide_eq_jobs(product(n_out(1:2,1)),&
                        &var_size_without_par,n_div,n_div_max=n_div_max)
                    chckerr('')
                case (2)                                                        ! HELENA
                    ! divide equilibrium jobs
                    ierr = divide_eq_jobs(product(n_out(1:2,1)),&
                        &var_size_without_par,n_div,n_div_max=n_div_max,&
                        &n_par_x_base=nchi)
```

```
                    chckerr('')
            end select

            ! calculate equilibrium job limits
            ierr = calc_eq_jobs_lims(n_out(1,1),n_div)
            chckerr('')

            ! set up total number of parallel points
            n_par_tot = maxval(eq_jobs_lims(2,:))-minval(eq_jobs_lims(1,:))+1
        else
            allocate(eq_jobs_lims(2,0))
        end if

        ! Calculate variables  on HELENA output  grid if full output  and HELENA
        ! for later interpolation
        if (post_output_full .and. eq_style.eq.2) then
            ! user output
            call writo('Reconstructing HELENA output for later interpolation')
            call lvl_ud(1)

            ierr = reconstruct_pb3d_grid(grid_eq_hel,'eq',&
                &grid_limits=lims_norm(:,1))
            chckerr('')
            ierr = reconstruct_pb3d_grid(grid_x_hel,'X',&
                &grid_limits=lims_norm(:,2))
            chckerr('')
            ierr = reconstruct_pb3d_eq_2(grid_eq_hel,eq_2_hel,'eq_2')
            chckerr('')
            ierr = reconstruct_pb3d_x_1(mds_x,grid_x_hel,x_hel,'X_1')
            chckerr('')

            ! user output
            call lvl_ud(-1)
            call writo('HELENA output reconstructed for later interpolation')
        end if

        ! reconstruct normal part of full eq grid for XYZ reconstruction
        if (post_output_full .and. plot_grid_style.eq.0 .or. &
            &plot_grid_style.eq.3) then
            ! user output
            call writo('Reconstructing full eq grid for XYZ reconstruction')
            call lvl_ud(1)

            ierr = reconstruct_pb3d_grid(grid_eq_xyz,'eq',&
                &rich_lvl=rich_lvl_name,tot_rich=.true.,lim_pos=lim_loc)
            chckerr('')

            ! user output
            call lvl_ud(-1)
            call writo('Full eq grid reconstructed for XYZ reconstruction')
        end if

        ! user output
        call lvl_ud(-1)
        call writo('3-D plots prepared')

        ! clean up
        call grid_eq%dealloc()
        call grid_x%dealloc()
        call grid_sol%dealloc()
        call eq_1%dealloc()
    end function init_post

    !> The main driver routine for postprocessing.
    !!
    !! \note The PB3D output is given on different grids for different styles of
    !! the equilibrium model:
    !!  - VMEC: field-aligned grid on which EV problem has been solved.
    !!  - HELENA: output  grid  on  which equilibrium  and metric  variables are
    !!  tabulated.
    !!
    !! Furthermore,  if Richardson  extrapolation is  used, the  VMEC grids  and
    !! variables are contained in multiple HDF5 variables.
    !! These variables are needed here both  in a field-aligned and a plot grid.
    !!
    !! A consequence of  the above is that for VMEC  the field-aligned output is
    !! already given, but the output on the  plot grid has to be calculated from
    !! scratch, while for  HELENA both outputs have to be  interpolated from the
    !! output tables.
    !!
    !! The general workflow is as follows:
    !!  - take a subset of the output grids for the current equilibrium job.
    !!  - for <tt>POST_style = 1</tt> (extended grid):
    !!      * VMEC: recalculate variables
    !!      * HEL:  interpolate variables
    !!    for <tt>POST_style = 2</tt> (B-aligned grid):
    !!      * VMEC: read subset of variables
```

```
    !!      * HEL:  interpolate variables
    !!  - create helper variables
    !!  - create plots and outputs
    !!
    !! \return ierr
    integer function run_driver_post() result(ierr)
        use num_vars, only: no_output, no_plots, eq_style, eq_jobs_lims, &
            &eq_job_nr, plot_b, plot_j, plot_sol_xi, plot_sol_q, plot_kappa, &
            &post_output_sol, post_style, compare_tor_pos, plot_e_rec
        use pb3d_ops, only: reconstruct_pb3d_grid, reconstruct_pb3d_eq_2, &
            &reconstruct_pb3d_x_1
        use eq_ops, only: calc_eq, calc_t_hf, b_plot, j_plot, &
            &kappa_plot, delta_r_plot
        use eq_utilities, only: calc_inv_met
        use x_ops, only: calc_x
        use sol_ops, only: plot_sol_vec, decompose_energy
        use helena_ops, only: interp_hel_on_grid
        use num_utilities, only: calc_aux_utilities
#if ldebug
        use num_vars, only: ltest
#endif

        character(*), parameter :: rout_name = 'run_driver_POST'

        ! local variables
        type(grid_type) :: grids(3)                                         ! local output eq, X and sol grid, with limited parallel range and divided normal range
        type(eq_1_type) :: eq_1                                             ! flux equilibrium
        type(eq_2_type) :: eq_2                                             ! metric equilibrium
        type(x_1_type) :: x                                                 ! vectorial perturbation variables
        integer :: id, jd                                                   ! counter
        integer :: n_ev_out                                                 ! how many EV's are treated
        integer :: i_ev_out                                                 ! counter
        logical :: no_plots_loc                                             ! local copy of no_plots
        logical :: no_output_loc                                            ! local copy of no_output
        logical :: last_eq_job                                              ! last parallel job
        real(dp), allocatable :: xyz_eq(:,:,:,:)                            ! X, Y and Z on equilibrium output grid
        real(dp), allocatable :: xyz_sol(:,:,:,:)                           ! X, Y and Z on solution output grid
        complex(dp), allocatable :: sol_val_comp(:,:,:)                     ! solution eigenvalue for requested solutions
#if ldebug
        logical :: ltest_loc                                                ! local copy of ltest
#endif

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Parallel range for this job:')
        call lvl_ud(1)
        call writo(trim(i2str(eq_jobs_lims(1,eq_job_nr)))//'..'//&
            &trim(i2str(eq_jobs_lims(2,eq_job_nr)))//' of '//'1..'//&
            &trim(i2str(n_par_tot)))
        call lvl_ud(-1)

        ! some variables
        last_eq_job = eq_job_nr.eq.size(eq_jobs_lims,2)
        n_ev_out = sum(max_id-min_id+1)

        ! set up restricted output grids for this equilibrium job
        ierr = setup_out_grids(grids,xyz_eq,xyz_sol)
        chckerr('')

        ! set up output eq_1, eq_2 and X_1, depending on equilibrium style
        ierr = calc_eq(grids(1),eq_1)
        chckerr('')
        select case (eq_style)
            case (1)                                                        ! VMEC
                ! user output
                call writo('Calculate variables on output grid')
                call lvl_ud(1)

                ! back up no_plots and no_output and set to .true.
                no_plots_loc = no_plots; no_plots = .true.
                no_output_loc = no_output; no_output = .true.
#if ldebug
                ltest_loc = ltest; ltest = .false.
#endif

                ! Calculate the metric equilibrium quantitities
                ierr = calc_eq(grids(1),eq_1,eq_2)
                chckerr('')

                ! calculate X variables, vector phase
                ierr = calc_x(mds_x,grids(1),grids(2),eq_1,eq_2,x)
                chckerr('')

```

```
                ! reset no_plots and no_output
                no_plots = no_plots_loc
                no_output = no_output_loc
#if ldebug
                ltest = ltest_loc
#endif

                ! user output
                call lvl_ud(-1)
                call writo('Variables calculated on output grid')
            case (2)                                                        ! HELENA
                ! user output
                call writo('Interpolate variables angularly on output grid')
                call lvl_ud(1)

                call eq_2%init(grids(1),setup_e=.true.,setup_f=.true.)
                call x%init(mds_x,grids(2))
                ierr = interp_hel_on_grid(grid_eq_hel,grids(1),&
                    &eq_2=eq_2_hel,eq_2_out=eq_2,eq_1=eq_1,&
                    &grid_name='output equilibrium grid')
                chckerr('')
                ierr = interp_hel_on_grid(grid_x_hel,grids(2),x_1=x_hel,&
                    &x_1_out=x,grid_name='output perturbation grid')
                chckerr('')

                ! also set up transformation matrices for plots
                ierr = calc_t_hf(grids(1),eq_1,eq_2,[0,0,0])
                chckerr('')
                ierr = calc_inv_met(eq_2%T_FE,eq_2%T_EF,[0,0,0])
                chckerr('')

                ! user output
                call lvl_ud(-1)
                call writo('Variables interpolated angularly on output grid')
        end select

        ! user output
        call writo('Initialize 3-D plots')
        call lvl_ud(1)

        ! initialize  integrated energies and  open decompose log file  if first
        ! equilibrium job and energy reconstruction requested
        if (plot_e_rec .and. eq_job_nr.eq.1) then
            allocate(e_pot_int(7,n_ev_out))
            allocate(e_kin_int(2,n_ev_out))
            e_pot_int = 0._dp
            e_kin_int = 0._dp

            ierr = open_decomp_log()
            chckerr('')
        end if

        ! user output
        call lvl_ud(-1)
        call writo('3-D plots initialized')

        ! user output
        call writo('3-D PB3D output plots')
        call lvl_ud(1)

        ! plot displacement vector if comparing toroidal positions
        if (compare_tor_pos) then
            call writo('Plot the displacement vector')
            call lvl_ud(1)
            ierr = delta_r_plot(grids(1),eq_1,eq_2,xyz_eq)
            chckerr('')
            call lvl_ud(-1)
        end if

        ! plot magnetic field if requested
        if (plot_b) then
            call writo('Plot the magnetic field')
            call lvl_ud(1)
            ierr = b_plot(grids(1),eq_1,eq_2,plot_fluxes=post_style.eq.1,&
                &xyz=xyz_eq)
            chckerr('')
            call lvl_ud(-1)
        end if

        ! plot current if requested
        if (plot_j) then
            call writo('Plot the current')
            call lvl_ud(1)
            ierr = j_plot(grids(1),eq_1,eq_2,plot_fluxes=post_style.eq.1,&
                &xyz=xyz_eq)
            chckerr('')
            call lvl_ud(-1)
```

```
        end if

        ! plot magnetic field if requested
        if (plot_kappa) then
            call writo('Plot the curvature')
            call lvl_ud(1)
            ierr = kappa_plot(grids(1),eq_1,eq_2,xyz=xyz_eq)
            chckerr('')
            call lvl_ud(-1)
        end if

        if (post_output_sol) then
            ! user output
            call writo('Solution plots for different ranges')
            call lvl_ud(1)

            ! loop over three ranges
            i_ev_out = 1
            if (last_eq_job) allocate(sol_val_comp(2,2,n_ev_out))
            do jd = 1,3
                if (min_id(jd).le.max_id(jd)) call &
                    &writo('RANGE '//trim(i2str(jd))//': modes '//&
                    &trim(i2str(min_id(jd)))//'..'//trim(i2str(max_id(jd))))
                call lvl_ud(1)

                ! indices in each range
                do id = min_id(jd),max_id(jd)
                    ! user output
                    call writo('Mode '//trim(i2str(id))//'/'//&
                        &trim(i2str(size(sol%val)))//', with eigenvalue '&
                        &//trim(c2strt(sol%val(id))))
                    call lvl_ud(1)

                    ! plot solution vector
                    ierr = plot_sol_vec(mds_x,mds_sol,grids(1),grids(2),&
                        &grids(3),eq_1,eq_2,x,sol,xyz_sol,id,&
                        &[plot_sol_xi,plot_sol_q])
                    chckerr('')

                    if (plot_e_rec) then
                        ! user output
                        call writo('Decompose the energy into its terms')
                        call lvl_ud(1)
                        ierr = decompose_energy(mds_x,mds_sol,grids(1),grids(2),&
                            &grids(3),eq_1,eq_2,x,sol,vac,id,&
                            &b_aligned=post_style.eq.2,xyz=xyz_sol,&
                            &e_pot_int=e_pot_int(:,i_ev_out),&
                            &e_kin_int=e_kin_int(:,i_ev_out))
                        chckerr('')
                        call lvl_ud(-1)

                        ! write  to   log  file  and  save   difference  between
                        ! Eigenvalue and energy fraction if last parallel job
                        if (last_eq_job) then
                            ierr = write_decomp_log(id,e_pot_int(:,i_ev_out),&
                                &e_kin_int(:,i_ev_out))
                            chckerr('')

                            sol_val_comp(:,1,i_ev_out) = id*1._dp
                            sol_val_comp(:,2,i_ev_out) = [sol%val(id),&
                                &sum(e_pot_int(:,i_ev_out))/&
                                &sum(e_kin_int(:,i_ev_out))]
                        end if

                        ! increment counter
                        i_ev_out = i_ev_out + 1
                    end if

                    ! user output
                    call lvl_ud(-1)
                    call writo('Mode '//trim(i2str(id))//'/'//&
                        &trim(i2str(size(sol%val)))//' finished')
                end do

                call lvl_ud(-1)
                if (min_id(jd).le.max_id(jd)) call &
                    &writo('RANGE '//trim(i2str(jd))//' plotted')
            end do

            ! plot  difference between  Eigenvalue and  energy fraction  if last
            ! parallel job
            if (last_eq_job) then
                call plot_sol_val_comp(sol_val_comp)
            end if

            ! user output
            call lvl_ud(-1)
```

```
            call writo('Solution for different ranges plotted')
        end if

        ! user output
        call lvl_ud(-1)
        call writo('3-D output plots done')

        ! clean up
        call writo('Clean up')
        call lvl_ud(1)
        call grids(1)%dealloc()
        call grids(2)%dealloc()
        call grids(3)%dealloc()
        call eq_1%dealloc()
        call eq_2%dealloc()
        call x%dealloc()
        call lvl_ud(-1)
    end function run_driver_post

    !> Cleans up main driver for postprocessing.
    subroutine stop_post()
        use num_vars, only: plot_grid_style, eq_style, post_output_full, &
            &post_output_sol
        use input_utilities, only: dealloc_in

        call dealloc_in()
        if (post_output_sol) then
            call sol%dealloc()
            call vac%dealloc()
        end if
        if (post_output_full .and. eq_style.eq.2) then
            call grid_eq_hel%dealloc()
            call grid_x_hel%dealloc()
            call eq_2_hel%dealloc()
            call x_hel%dealloc()
        end if
        if (post_output_full .and. plot_grid_style.eq.0 .or. &
            &plot_grid_style.eq.3) then
            call grid_eq_xyz%dealloc()
        end if
    end subroutine stop_post

    !> Opens the decomposition log file.
    !!
    !! \return ierr
    integer function open_decomp_log() result(ierr)
        use num_vars, only: prog_name, output_name, rank, post_style, &
            &decomp_i
        use rich_vars, only: rich_lvl

        character(*), parameter :: rout_name = 'open_decomp_log'

        ! local variables
        character(len=max_str_ln) :: format_head                    ! header format
        character(len=max_str_ln) :: full_output_name               ! full name
        character(len=max_str_ln) :: grid_name                      ! name of grid on which calculations are done
        character(len=2*max_str_ln) :: temp_output_str              ! temporary output string

        ! initialize ierr
        ierr = 0

        ! set up output name
        select case (post_style)
            case (1)                                                ! extended grid
                grid_name = 'extended'
            case (2)                                                ! field-aligned grid
                grid_name = 'field-aligned'
        end select

        call writo('Open decomposition log file')
        call lvl_ud(1)
        if (rank.eq.0) then
            ! set format strings
            format_head = '("#  ",7(A23," "))'
            ! open output file for the log
            full_output_name = prog_name//'_'//trim(output_name)//'_EN_R'//&
                &trim(i2str(rich_lvl))//'.txt'
            open(unit=decomp_i,status='replace',file=full_output_name,&
                &iostat=ierr)
            chckerr('Cannot open EN output file')
            call writo('Log file opened in '//trim(full_output_name))
            write(unit=decomp_i,fmt='(A)',iostat=ierr) &
                &'# Energy decomposition using the solution Eigenvectors'
            chckerr('Failed to write')
            write(unit=decomp_i,fmt='(A)',iostat=ierr) &
                &'# (Output on the '//trim(grid_name)//' grid)'
```

```
            chckerr('Failed to write')
            write(temp_output_str,format_head) &
                &'RE Eigenvalue          ', 'RE E_pot/E_kin         ', &
                &'RE E_pot               ', 'RE E_kin               '
            write(unit=decomp_i,fmt='(A)',iostat=ierr) trim(temp_output_str)
            chckerr('Failed to write')
            write(temp_output_str,format_head) &
                &'IM Eigenvalue          ', 'IM E_pot/E_kin         ', &
                &'IM E_pot               ', 'IM E_kin               '
            write(unit=decomp_i,fmt='(A)',iostat=ierr) trim(temp_output_str)
            chckerr('Failed to write')
            write(temp_output_str,format_head) &
                &'RE E_kin_n             ', 'RE E_kin_g             '
            write(unit=decomp_i,fmt='(A)',iostat=ierr) trim(temp_output_str)
            chckerr('Failed to write')
            write(temp_output_str,format_head) &
                &'IM E_kin_n             ', 'IM E_kin_g             '
            write(unit=decomp_i,fmt='(A)',iostat=ierr) trim(temp_output_str)
            chckerr('Failed to write')
            write(temp_output_str,format_head) &
                &'RE E_pot line_bending_n', 'RE E_pot line_bending_g', &
                &'RE E_pot ballooning_n  ', 'RE E_pot ballooning_g  ', &
                &'RE E_pot kink_n        ', 'RE E_pot kink_g        ', &
                &'RE E_pot vac           '
            write(unit=decomp_i,fmt='(A)',iostat=ierr) trim(temp_output_str)
            chckerr('Failed to write')
            write(temp_output_str,format_head) &
                &'IM E_pot line_bending_n', 'IM E_pot line_bending_g', &
                &'IM E_pot ballooning_n  ', 'IM E_pot ballooning_g  ', &
                &'IM E_pot kink_n        ', 'IM E_pot kink_g        ', &
                &'IM E_pot vac           '
            write(unit=decomp_i,fmt='(A)',iostat=ierr) trim(temp_output_str)
            chckerr('Failed to write')
        end if
        call lvl_ud(-1)
    end function open_decomp_log

    !> Write to decomposition log file.
    !!
    !! \return ierr
    integer function write_decomp_log(X_id,E_pot_int,E_kin_int) result(ierr)
        use num_vars, only: decomp_i, rank

        character(*), parameter :: rout_name = 'write_decomp_log'

        ! input / output
        integer, intent(in) :: x_id                                     !< nr. of Eigenvalue
        complex(dp), intent(in) :: e_pot_int(7)                         !< E_pot integrated for requested solutions
        complex(dp), intent(in) :: e_kin_int(2)                         !< E_kin integrated for requested solutions

        ! local variables
        character(len=max_str_ln) :: format_val                         ! format
        character(len=2*max_str_ln) :: temp_output_str                  ! temporary output string

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Write to log file')
        call lvl_ud(1)

        ! only master
        if (rank.eq.0) then
            ! set up format string:
            !   row 1: EV, E_pot/E_kin
            !   row 2: E_pot, E_kin
            !   row 3: E_kin(1), E_kin(2)
            !   row 4: E_pot(1), E_pot(2)
            !   row 5: E_pot(3), E_pot(4)
            !   row 6: E_pot(5), E_pot(6)
            format_val = '("  ",7(ES23.16," "))'

            ! write header
            write(unit=decomp_i,fmt='(A)',iostat=ierr) &
                &'# Eigenvalue '//trim(i2str(x_id))
            chckerr('Failed to write')

            ! write values
            write(temp_output_str,format_val) &
                &rp(sol%val(x_id)),&
                &rp(sum(e_pot_int)/sum(e_kin_int)),&
                &rp(sum(e_pot_int)),&
                &rp(sum(e_kin_int))
            write(unit=decomp_i,fmt='(A)',iostat=ierr) &
                &trim(temp_output_str)
            chckerr('Failed to write')
            write(temp_output_str,format_val) &
```

```
                &ip(sol%val(x_id)),&
                &ip(sum(e_pot_int)/sum(e_kin_int)), &
                &ip(sum(e_pot_int)),&
                &ip(sum(e_kin_int))
            write(unit=decomp_i,fmt='(A)',iostat=ierr) &
                &trim(temp_output_str)
            chckerr('Failed to write')
            write(temp_output_str,format_val) &
                &rp(e_kin_int(1)),&
                &rp(e_kin_int(2))
            write(unit=decomp_i,fmt='(A)',iostat=ierr) &
                &trim(temp_output_str)
            chckerr('Failed to write')
            write(temp_output_str,format_val) &
                &ip(e_kin_int(1)),&
                &ip(e_kin_int(2))
            write(unit=decomp_i,fmt='(A)',iostat=ierr) &
                &trim(temp_output_str)
            chckerr('Failed to write')
            write(temp_output_str,format_val) &
                &rp(e_pot_int(1)),&
                &rp(e_pot_int(2)),&
                &rp(e_pot_int(3)),&
                &rp(e_pot_int(4)),&
                &rp(e_pot_int(5)),&
                &rp(e_pot_int(6)),&
                &rp(e_pot_int(7))
            write(unit=decomp_i,fmt='(A)',iostat=ierr) &
                &trim(temp_output_str)
            chckerr('Failed to write')
            write(temp_output_str,format_val) &
                &ip(e_pot_int(1)),&
                &ip(e_pot_int(2)),&
                &ip(e_pot_int(3)),&
                &ip(e_pot_int(4)),&
                &ip(e_pot_int(5)),&
                &ip(e_pot_int(6)),&
                &ip(e_pot_int(7))
            write(unit=decomp_i,fmt='(A)',iostat=ierr) &
                &trim(temp_output_str)
            chckerr('Failed to write')

            close(decomp_i)
        end if

        call lvl_ud(-1)
    end function write_decomp_log

    !> finds the plot ranges \c min_id and \c max_id.
    !!
    !! There are three ranges, calculated using n_sol_plotted, which indicates:
    !!  1. how many of the first EV's in the unstable range
    !!  2. how many of the last EV's in the unstable range
    !!  3. how many of the first EV's in the stable range
    !!  4. how many of the last EV's in the stable range
    !!
    !! have to  be plotted.
    !!
    !! This yields maximally  three different ranges: One starting  at the first
    !! unstable EV, one centered  around the zero of the EV's  and one ending at
    !! the last EV. These ranges can be disjoint but do not have to be.\n
    !! Also, it is  possible that a range  does not exist, for  example if there
    !! are no unstable EV's.
    !!
    !! \note  A negative  value for  the  elements in  n_sol_plotted means  "all
    !! values in range":
    !!  1. or 2. full unstable range
    !!  3. or 4. full stable range
    subroutine find_stab_ranges(sol,min_id,max_id,last_unstable_id)
        use num_vars, only: n_sol_plotted

        ! input / output
        type(sol_type), intent(in) :: sol                                   !< solution variables
        integer, intent(inout) :: min_id(3)                                 !< min. index of range 1, 2 and 3
        integer, intent(inout) :: max_id(3)                                 !< max. index of range 1, 2 and 3
        integer, intent(inout) :: last_unstable_id                          !< index of last unstable EV

        ! local variables
        integer :: id                                                       ! counter
        integer :: n_sol_found                                              ! how many solutions found and saved

        ! set local variables
        n_sol_found = size(sol%val)

        ! find last unstable index (if ends with 0, no unstable EV)
        last_unstable_id = 0
        do id = 1,n_sol_found
```

```
            if (rp(sol%val(id)).lt.0._dp) last_unstable_id = id
        end do
        ! set up min. and max. of range 1
        if (last_unstable_id.gt.0) then                                  ! there is an unstable range
            if (n_sol_plotted(1).ge.0) then
                min_id(1) = 1                                            ! start from most unstable EV
                max_id(1) = min(n_sol_plotted(1),last_unstable_id)       ! end with n_sol_plotted(1) first unstable values if available
            else
                min_id(1) = 1                                            ! start from most unstable EV
                max_id(1) = last_unstable_id                             ! end with last unstable EV
            end if
        else                                                             ! no unstable range
            min_id(1) = 1                                                ! no unstable values to plot
            max_id(1) = 0                                                ! no unstable values to plot
        end if
        ! set up min. and max. of range 2
        if (last_unstable_id.gt.0) then                                  ! there is an unstable range
            if (n_sol_plotted(2).ge.0) then
                min_id(2) = last_unstable_id - n_sol_plotted(2) + 1      ! start from n_sol_plotted(2) last unstable values
            else
                min_id(2) = 1                                            ! start from 1
            end if
            if (n_sol_plotted(3).ge.0) then
                max_id(2) = min(last_unstable_id + n_sol_plotted(3),n_sol_found)! end with n_sol_plotted(3) first stable values if available
            else
                max_id(2) = n_sol_found                                  ! end with last EV
            end if
        else                                                             ! no unstable range
            min_id(2) = 1                                                ! start from first EV (stable)
            if (n_sol_plotted(3).ge.0) then
                max_id(2) = min(n_sol_plotted(3),n_sol_found)            ! end with n_sol_plotted(3) first stable values if available
            else
                max_id(2) = n_sol_found                                  ! end with last EV
            end if
        end if
        ! set up min. and max. of range 3
        if (n_sol_plotted(4).ge.0) then
            min_id(3) = n_sol_found - n_sol_plotted(4) + 1               ! start from n_sol_plotted(4) last stable values
        else
            min_id(3) = last_unstable_id + 1                             ! start from n_sol_plotted(4) last stable values
        end if
        max_id(3) = n_sol_found                                          ! end with most stable EV
        ! merge ranges 2 and 3 if overlap
        if (min_id(3).le.max_id(2)) then
            max_id(2) = max_id(3)                                        ! range 3 merged with range 2
            min_id(3) = 1                                                ! range 3 not used any more
            max_id(3) = 0                                                ! range 3 not used any more
        end if
        ! merge ranges 1 and 2 if overlap
        if (min_id(2).le.max_id(1)) then
            max_id(1) = max_id(2)                                        ! range 2 merged with range 1
            min_id(2) = 1                                                ! range 2 not used any more
            max_id(2) = 0                                                ! range 2 not used any more
        end if
    end subroutine find_stab_ranges

    !> Plots difference between Eigenvalues and energy fraction.
    subroutine plot_sol_val_comp(sol_val_comp)
        use num_vars, only: rank

        ! input /output
        complex(dp), intent(inout) :: sol_val_comp(:,:,:)               !< fraction between total \f$E_{\text{pot}}\f$ and \f$E_{\text{kin}}\f$, compared with EV

        ! local variables
        character(len=max_str_ln) :: plot_title(2)                       ! title for plots
        character(len=max_str_ln) :: plot_name                           ! file name for plots

        if (rank.eq.0 .and. size(sol_val_comp,3).gt.0) then
            ! real part
            plot_title = ['RE sol_val','E_frac    ']
            plot_name = 'sol_val_comp_RE'
            call print_ex_2d(plot_title,plot_name,&
                &rp(sol_val_comp(:,1,:)),&
                &x=rp(sol_val_comp(:,2,:)),draw=.false.)
            call draw_ex(plot_title,plot_name,size(sol_val_comp,3),1,.false.)
            plot_title(1) = 'rel diff between RE sol_val and E_frac'
            plot_name = 'sol_val_comp_RE_rel_diff'
            call print_ex_2d(plot_title(1),plot_name,rp(&
                &(sol_val_comp(1,2,:)-sol_val_comp(2,2,:))/&
                &sol_val_comp(1,2,:)),x=rp(sol_val_comp(1,1,:)),&
```

```
                &draw=.false.)
            call draw_ex(plot_title(1:1),plot_name,1,1,.false.)
            plot_title(1) = 'rel diff between RE sol_val and E_frac [log]'
            plot_name = 'sol_val_comp_RE_rel_diff_log'
            call print_ex_2d(plot_title(1),plot_name,log10(abs(rp(&
                &(sol_val_comp(1,2,:)-sol_val_comp(2,2,:))/&
                &sol_val_comp(1,2,:)))),x=rp(sol_val_comp(1,1,:)),&
                &draw=.false.)
            call draw_ex(plot_title(1:1),plot_name,1,1,.false.)

            ! imaginary part
            plot_title = ['IM sol_val','E_frac    ']
            plot_name = 'sol_val_comp_IM'
            call print_ex_2d(plot_title,plot_name,&
                &rp(sol_val_comp(:,1,:)),x=ip(sol_val_comp(:,2,:)),draw=.false.)
            call draw_ex(plot_title,plot_name,size(sol_val_comp,3),1,.false.)
            plot_title(1) = 'rel diff between IM sol_val and E_frac'
            plot_name = 'sol_val_comp_IM_rel_diff'
            call print_ex_2d(plot_title(1),plot_name,ip(&
                &(sol_val_comp(1,2,:)-sol_val_comp(2,2,:))/&
                &sol_val_comp(1,2,:)),x=rp(sol_val_comp(1,1,:)),&
                &draw=.false.)
            call draw_ex(plot_title(1:1),plot_name,1,1,.false.)
            plot_title(1) = 'rel diff between IM sol_val and E_frac [log]'
            plot_name = 'sol_val_comp_IM_rel_diff_log'
            call print_ex_2d(plot_title(1),plot_name,log10(abs(ip(&
                &(sol_val_comp(1,2,:)-sol_val_comp(2,2,:))/&
                &sol_val_comp(1,2,:)))),x=rp(sol_val_comp(1,1,:)),&
                &draw=.false.)
            call draw_ex(plot_title(1:1),plot_name,1,1,.false.)
        end if
    end subroutine plot_sol_val_comp

    !> Sets up the output grids for a particular parallel job.
    !!
    !! Three grids are returned:
    !!  * eq
    !!  * X
    !!  * sol
    !!
    !! The normal coordinates of these grids correspond to the one used in PB3D.
    !! For X this is determined by \c x_grid_style.
    !!
    !! The  angular  components of  the  eq  and X  grids  is  determined by  \c
    !! post_style:
    !!  * 1:  extended  grid  with  \c  min_theta_plot,  \c  max_theta_plot,  \c
    !!  min_zeta_plot and \c max_zeta_plot.
    !!  * 2: PB3D grid.
    !!
    !! \return ierr
    integer function setup_out_grids(grids_out,XYZ_eq,XYZ_sol) result(ierr)
        use num_vars, only: min_theta_plot, max_theta_plot, post_style, &
            &eq_job_nr, eq_jobs_lims, norm_disc_prec_x, x_grid_style
        use num_utilities, only: spline
        use grid_utilities, only: extend_grid_f
        use pb3d_ops, only: reconstruct_pb3d_grid
        use num_vars, only: rank, n_procs
#if ldebug
        use grid_utilities, only: nufft
#endif

        character(*), parameter :: rout_name = 'setup_out_grids'

        ! input / output
        type(grid_type), intent(inout) :: grids_out(3)                          !< eq, X and sol output grids (full parallel and normal range)
        real(dp), intent(inout), allocatable :: xyz_eq(:,:,:,:)                 !< \c X, \c Y and \c Z on output equilibrium grid
        real(dp), intent(inout), allocatable :: xyz_sol(:,:,:,:)                  !< \c X, \c Y and \c Z on output solution grid

        ! local variables
        type(grid_type) :: grid_eq                                              ! equilibrium grid
        type(grid_type) :: grid_x                                               ! perturbation grid
        integer :: n_theta                                                      ! nr. of points in theta for extended grid
        integer :: id, jd, ld                                                   ! counters
        integer :: lim_loc(3,2,3)                                               ! grid ranges for local equilibrium job (last index: eq, X, sol)
        real(dp) :: lim_theta(2)                                                ! theta limits
        real(dp), allocatable :: r_geo(:,:,:)                                   ! geometrical radius
        real(dp), allocatable :: xy(:,:,:)                                      ! x and y
        real(dp), allocatable :: f(:,:,:)                                       ! Fourier components
        real(dp), allocatable :: f_loc(:,:)                                     ! local f
        real(dp), allocatable :: xyz_x(:,:,:,:)                                 ! X, Y and Z on output perturbation grid

        ! initialize ierr
        ierr = 0
```

```

        call writo('Set up output grid')
        call lvl_ud(1)

        select case (post_style)
            case (1)                                                        ! extended grid
                ! limits to exclude angular part
                call writo('Set up limits')
                ! eq
                lim_loc(1,:,1) = [0,0]
                lim_loc(2,:,1) = [0,0]
                lim_loc(3,:,1) = [1,-1]
                ! X
                lim_loc(1,:,2) = [0,0]
                lim_loc(2,:,2) = [0,0]
                lim_loc(3,:,2) = [1,-1]
                ! sol
                lim_loc(1,:,3) = [0,0]
                lim_loc(2,:,3) = [0,0]
                lim_loc(3,:,3) = [1,-1]

                ! reconstruct equilibrium, perturbation and solution grids
                call writo('Reconstruct PB3D variables')
                ierr = reconstruct_pb3d_grid(grid_eq,'eq',&
                    &rich_lvl=rich_lvl_name,lim_pos=lim_loc(:,:,1),&
                    &grid_limits=lims_norm(:,1))
                chckerr('')
                ierr = reconstruct_pb3d_grid(grid_x,'X',&
                    &rich_lvl=rich_lvl_name,lim_pos=lim_loc(:,:,2),&
                    &grid_limits=lims_norm(:,2))
                chckerr('')
                ierr = reconstruct_pb3d_grid(grids_out(3),'sol',&
                    &lim_pos=lim_loc(:,:,3),grid_limits=lims_norm(:,3))
                chckerr('')

                ! theta limits
                if (n_par_tot.gt.1) then
                    lim_theta = min_theta_plot + &
                        &(eq_jobs_lims(:,eq_job_nr)-1._dp)*&
                        &(max_theta_plot-min_theta_plot)/(n_par_tot-1._dp)
                else
                    lim_theta = min_theta_plot
                end if
                n_theta = eq_jobs_lims(2,eq_job_nr)-eq_jobs_lims(1,eq_job_nr)+1

                ! extend eq and X grid
                call writo('Extend equilibrium grid')
                call lvl_ud(1)
                ierr = extend_grid_f(grid_eq,grids_out(1),n_theta_plot=n_theta,&
                    &lim_theta_plot=lim_theta,grid_eq=grid_eq)              ! extend eq grid and convert to F
                chckerr('')
                call lvl_ud(-1)

                call writo('Extend perturbation grid')
                call lvl_ud(1)
                ierr = grids_out(2)%init([n_theta,n_out(2:3,2)],&
                    &i_lim=[grid_x%i_min,grid_x%i_max])
                chckerr('')
                grids_out(2)%r_F = grid_x%r_F
                grids_out(2)%r_E = grid_x%r_E
                grids_out(2)%loc_r_F = grid_x%loc_r_F
                grids_out(2)%loc_r_E = grid_x%loc_r_E
                select case (x_grid_style)
                    case (1)                                                ! equilibrium
                        ! copy angular variables
                        grids_out(2)%theta_F = grids_out(1)%theta_F
                        grids_out(2)%theta_E = grids_out(1)%theta_E
                        grids_out(2)%zeta_F = grids_out(1)%zeta_F
                        grids_out(2)%zeta_E = grids_out(1)%zeta_E
                    case (2,3)                                              ! solution or enriched
                        ! interpolate angular variables
                        do jd = 1,grids_out(1)%n(2)
                            do id = 1,grids_out(1)%n(1)
                                ierr = spline(grids_out(1)%loc_r_F,&
                                    &grids_out(1)%theta_F(id,jd,:),&
                                    &grids_out(2)%loc_r_F,&
                                    &grids_out(2)%theta_F(id,jd,:),&
                                    &ord=norm_disc_prec_x)
                                chckerr('')
                                ierr = spline(grids_out(1)%loc_r_F,&
                                    &grids_out(1)%theta_E(id,jd,:),&
                                    &grids_out(2)%loc_r_F,&
                                    &grids_out(2)%theta_E(id,jd,:),&
                                    &ord=norm_disc_prec_x)
                                chckerr('')
                                ierr = spline(grids_out(1)%loc_r_F,&
                                    &grids_out(1)%zeta_F(id,jd,:),&
```

```
                                    &grids_out(2)%loc_r_F,&
                                    &grids_out(2)%zeta_F(id,jd,:),&
                                    &ord=norm_disc_prec_x)
                                chckerr('')
                                ierr = spline(grids_out(1)%loc_r_F,&
                                    &grids_out(1)%zeta_E(id,jd,:),&
                                    &grids_out(2)%loc_r_F,&
                                    &grids_out(2)%zeta_E(id,jd,:),&
                                    &ord=norm_disc_prec_x)
                                chckerr('')
                            end do
                        end do
                    end select
                    call lvl_ud(-1)

                    ! clean up
                    call grid_eq%dealloc()
                    call grid_x%dealloc()
                case (2)                                                    ! field-aligned grid
                    ! limits to take subset of angular part
                    call writo('Set up limits')
                    ! eq
                    lim_loc(1,:,1) = eq_jobs_lims(:,eq_job_nr)
                    lim_loc(2,:,1) = [1,-1]
                    lim_loc(3,:,1) = [1,-1]
                    ! X
                    lim_loc(1,:,2) = eq_jobs_lims(:,eq_job_nr)
                    lim_loc(2,:,2) = [1,-1]
                    lim_loc(3,:,2) = [1,-1]
                    ! sol
                    lim_loc(1,:,3) = [0,0]
                    lim_loc(2,:,3) = [0,0]
                    lim_loc(3,:,3) = [1,-1]

                    ! reconstruct equilibrium, perturbation and solution grids
                    call writo('Reconstruct PB3D variables')
                    ierr = reconstruct_pb3d_grid(grids_out(1),trim(grid_name(1)),&
                        &rich_lvl=rich_lvl,tot_rich=.true.,&
                        &lim_pos=lim_loc(:,:,1),grid_limits=lims_norm(:,1))
                    chckerr('')
                    ierr = reconstruct_pb3d_grid(grids_out(2),trim(grid_name(2)),&
                        &rich_lvl=rich_lvl,tot_rich=.true.,&
                        &lim_pos=lim_loc(:,:,2),grid_limits=lims_norm(:,2))
                    chckerr('')
                    ierr = reconstruct_pb3d_grid(grids_out(3),trim(grid_name(3)),&
                        &lim_pos=lim_loc(:,:,3),grid_limits=lims_norm(:,3))
                    chckerr('')
            end select

            call writo('Calculate X, Y and Z of equilibrium grid')
            call lvl_ud(1)
            ierr = calc_xyz_of_output_grid(grids_out(1),xyz_eq)
            chckerr('')
            call lvl_ud(-1)

            call writo('Calculate X, Y and Z of perturbation grid')
            call lvl_ud(1)
            ierr = calc_xyz_of_output_grid(grids_out(2),xyz_x)
            chckerr('')
            call lvl_ud(-1)

            call writo('Calculate X, Y and Z of solution grid')
            call lvl_ud(1)
            allocate(xyz_sol(grids_out(2)%n(1),grids_out(2)%n(2),&
                &grids_out(3)%loc_n_r,3))
            select case (x_grid_style)
                case (1,3)                                                  ! equilibrium or enriched
                    ! setup normal interpolation data
                    do ld = 1,3
                        do jd = 1,grids_out(2)%n(2)
                            do id = 1,grids_out(2)%n(1)
                                ierr = spline(grids_out(2)%loc_r_F,&
                                    &xyz_x(id,jd,:,ld),grids_out(3)%loc_r_F,&
                                    &xyz_sol(id,jd,:,ld),ord=norm_disc_prec_x)
                                chckerr('')
                            end do
                        end do
                    end do
                case (2)                                                    ! solution
                    xyz_sol = xyz_x
            end select
            call lvl_ud(-1)

#if ldebug
            ! Plot toroidal dependency of ripple.
            ! Note: Hard-code NFP.
            if (debug_tor_dep_ripple .and. rank.eq.n_procs-1) then
```

```
            call plot_hdf5(['X','Y','Z'],'XYZ',xyz_eq)
            allocate(r_geo(grids_out(1)%n(1),grids_out(1)%n(2),&
                &grids_out(1)%loc_n_r))
            r_geo = sqrt((xyz_eq(:,:,:,3)-0.56710)**2 + &
                &(xyz_eq(:,:,:,1)-6.4297)**2)
            call plot_hdf5('r_geo','r_geo_TEMP',r_geo,x=xyz_eq(:,:,:,1),&
                &y=xyz_eq(:,:,:,2),z=xyz_eq(:,:,:,3))
            allocate(xy(grids_out(1)%n(2),grids_out(1)%n(1),2))
            do id = 1,grids_out(1)%n(1)
                xy(:,id,1) = xyz_eq(id,:,grids_out(1)%loc_n_r,2)*18
                xy(:,id,2) = r_geo(id,:,grids_out(1)%loc_n_r) &
                    &* grids_out(1)%n(2) / sum(r_geo(id,:,grids_out(1)%loc_n_r))
            end do
            call print_ex_2d(['r_geo'],'r_geo',xy(:,:,2),x=xy(:,:,1),&
                &draw=.false.)
            call draw_ex(['r_geo'],'r_geo',grids_out(1)%n(1),1,.false.,&
                &ex_plot_style=1)
            do id = 1,grids_out(1)%n(1)
                ierr = nufft(xy(1:size(xy,1)-1,id,1),xy(1:size(xy,1)-1,id,2),&
                    &f_loc)
                chckerr('')
                if (id.eq.1) &
                    &allocate(f(size(f_loc,1),size(f_loc,2),grids_out(1)%n(1)))
                f(:,:,id) = f_loc
            end do
            call print_ex_2d(['r_geo_cos'],'r_geo_cos',f(:,1,:),&
                &draw=.false.)
            call draw_ex(['r_geo_cos'],'r_geo_cos',grids_out(1)%n(1),1,.false.,&
                &ex_plot_style=1)
            call print_ex_2d(['r_geo_sin'],'r_geo_sin',f(:,2,:),&
                &draw=.false.)
            call draw_ex(['r_geo_sin'],'r_geo_sin',grids_out(1)%n(1),1,.false.,&
                &ex_plot_style=1)
        end if
#endif

        call lvl_ud(-1)
    contains
        ! Calculate XYZ of output grid, depending on plot_grid style:
        !   0: 3-D geometry
        !   1: slab geometry
        !   2: slab geometry with wrapping of angles
        !   3: 3-D geometry with straightened toroidal coordinate
        !> \private
        integer function calc_xyz_of_output_grid(grid,XYZ) result(ierr)
            use grid_utilities, only: calc_xyz_grid
            use num_vars, only: eq_style, plot_grid_style, swap_angles, &
                &use_pol_flux_f
            use eq_vars, only: max_flux_f
            use grid_vars, only: alpha, n_alpha
            use vmec_utilities, onLy: calc_trigon_factors
            use mpi_utilities, only: wait_mpi

            character(*), parameter :: rout_name = 'calc_XYZ_of_output_grid'

            ! input / output
            type(grid_type), intent(inout) :: grid                          ! output grid for which to calculate XYZ
            real(dp), allocatable :: xyz(:,:,:,:)                           ! X, Y and Z on output grid

            ! local variables
            real(dp), allocatable :: r(:,:,:)                               ! R on output grid
            character(len=max_str_ln) :: err_msg                            ! error message
            integer :: jd, kd                                               ! counters

            ! initialize ierr
            ierr = 0

            ! if VMEC, calculate trigonometric factors of output grid
            if (eq_style.eq.1) then
                ierr = calc_trigon_factors(grid%theta_E,grid%zeta_E,&
                    &grid%trigon_factors)
                chckerr('')
            end if

            ! set up XYZ
            allocate(xyz(grid%n(1),grid%n(2),grid%loc_n_r,3))
            select case (plot_grid_style)
                case (0)                                                    ! 3-D geometry
                    ! user output
                    call writo('Plots are done in 3-D geometry')
                    call lvl_ud(1)

                    ierr = calc_xyz_grid(grid_eq_xyz,grid,&
                        &xyz(:,:,:,1),xyz(:,:,:,2),xyz(:,:,:,3))
                    chckerr('')
                    !!! To plot the cross-section
                    !!call print_ex_2D(['cross_section'],'cross_section_'//&
```

```
                        !!&trim(i2str(eq_job_nr))//'_'//trim(i2str(rank)),&
                        !!&XYZ(:,1,:,3),x=XYZ(:,1,:,1),draw=.false.)
                    !!call draw_ex(['cross_section'],'cross_section_'//&
                        !!&trim(i2str(eq_job_nr))//'_'//trim(i2str(rank)),&
                        !!&size(XYZ,3),1,.false.)

                    call lvl_ud(-1)
                case (3)                                                ! 3-D geometry with straightened toroidal coordinate
                    ! user output
                    call writo('Plots are done in an unwrapped torus')
                    call lvl_ud(1)

                    allocate(r(grid%n(1),grid%n(2),grid%loc_n_r))
                    ierr = calc_xyz_grid(grid_eq_xyz,grid,&
                        &xyz(:,:,:,1),xyz(:,:,:,2),xyz(:,:,:,3),r=r)
                    chckerr('')
                    xyz(:,:,:,1) = r                                    ! set X to R coordinate
                    xyz(:,:,:,2) = grid%zeta_F                          ! set Y to toroidal coordinate
                    !!! To plot the cross-section
                    !!call print_ex_2D(['cross_section'],'cross_section_'//&
                        !!&trim(i2str(eq_job_nr))//'_'//trim(i2str(rank)),&
                        !!&XYZ(:,1,:,3),x=XYZ(:,1,:,1),draw=.false.)
                    !!call draw_ex(['cross_section'],'cross_section_'//&
                        !!&trim(i2str(eq_job_nr))//'_'//trim(i2str(rank)),&
                        !!&size(XYZ,3),1,.false.)

                    call lvl_ud(-1)
                case (1,2)                                              ! slab geometry (with or without wrapping to fundamental interval)
                    ! user output
                    call writo('Plots are done in slab geometry')
                    call lvl_ud(1)

                    ! set angular coordinates
                    select case (post_style)
                        case (1)                                        ! extended grid: both theta and zeta need to be chosen, not alpha
                            if (use_pol_flux_f) then
                                xyz(:,:,:,1) = grid%theta_F/pi
                                xyz(:,:,:,2) = grid%zeta_F/pi
                            else
                                xyz(:,:,:,1) = grid%zeta_F/pi
                                xyz(:,:,:,2) = grid%theta_F/pi
                            end if
                        case (2)                                        ! field-aligned grid: alpha in combination with parallel angle
                            if (swap_angles) then
                                call writo('For plotting, the angular &
                                    &coordinates are swapped: theta <-> zeta')
                                if (use_pol_flux_f) then
                                    xyz(:,:,:,1) = grid%zeta_F/pi
                                else
                                    xyz(:,:,:,1) = grid%theta_F/pi
                                end if
                            else
                                if (use_pol_flux_f) then
                                    xyz(:,:,:,1) = grid%theta_F/pi
                                else
                                    xyz(:,:,:,1) = grid%zeta_F/pi
                                end if
                            end if
                            do jd = 1,n_alpha
                                xyz(:,jd,:,2) = alpha(jd)
                            end do
                    end select

                    ! set normal coordinate
                    do kd = 1,grid%loc_n_r
                        xyz(:,:,kd,3) = grid%loc_r_F(kd)/max_flux_f*2*pi
                    end do

                    ! limit to fundamental interval -1..1 if plot grid style 2
                    if (plot_grid_style.eq.2) xyz(:,:,:,1:2) = &
                        &modulo(xyz(:,:,:,1:2)+1._dp,2._dp)-1._dp

                    call lvl_ud(-1)
                case default
                    err_msg = 'No style "'//trim(i2str(plot_grid_style))//&
                        &'" for plot_grid_style'
                    ierr = 1
                    chckerr(err_msg)
            end select

            ! synchronize
            ierr = wait_mpi()
            chckerr('')
```

```
        end function calc_xyz_of_output_grid
    end function setup_out_grids
end module driver_post
```

D.12 /Users/toonweyens/Code/PB3D/Modules/driver_sol.f90 File Reference

Modules

- module driver_sol

  Driver of the solution part of PB3D.

Functions/Subroutines

- integer function, public driver_sol::run_driver_sol (grid_eq, grid_x, grid_sol, x, vac, sol)

  Main driver of PB3D solution part.
- integer function driver_sol::interp_v (mds_i, grid_i, x_i, mds_o, grid_o, x_o)

  Interpolate tensorial perturbation quantities in the third dimension.

D.13 driver_sol.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Driver of the solution part of PB3D.
!-----------------------------------------------------------------------------!
module driver_sol
#include <PB3D_macros.h>
#include <wrappers.h>
    use str_utilities
    use output_ops
    use messages
    use num_vars, only: dp, pi, max_str_ln, iu
    use grid_vars, only: grid_type
    use x_vars, only: x_2_type, modes_type, &
        &mds_x, mds_sol
    use vac_vars, only: vac_type
    use sol_vars, only: sol_type

    implicit none
    private
    public run_driver_sol

    ! global variables
#if ldebug
    logical :: debug_run_driver_sol = .false.                                   !< debug information for run_driver_sol \ldebug
    logical :: debug_interp_V = .false.                                         !< debug information for interp_v \ldebug
    integer :: ivs_stats(3)                                                     !< statistics of interp_v_spline (copy, linear, spline)
#endif

contains
    !> Main driver of PB3D solution part.
    !!
    !!  - sets up:
    !!      * \c grid_sol (only first Richardson level)
    !!      * \c sol
    !!  - writes to HDF5:
    !!      * \c grid_sol (only first Richardson level)
    !!      * \c sol
    !!  - deallocates:
    !!      * sol before setting up (but after guess)
    !!
```

```
    !! \return ierr
    integer function run_driver_sol(grid_eq,grid_X,grid_sol,X,vac,sol) &
        &result(ierr)

        use num_vars, only: ev_style, eq_style, rich_restart_lvl, rank, &
            &n_procs, x_grid_style, jump_to_sol
        use grid_vars, only: n_r_sol, n_alpha
        use pb3d_ops, only: reconstruct_pb3d_grid, reconstruct_pb3d_sol
        use slepc_ops, only: solve_ev_system_slepc
        use grid_ops, only: calc_norm_range, setup_grid_sol, &
            &print_output_grid, redistribute_output_grid
        use sol_ops, only: print_output_sol
        use rich_vars, only: rich_lvl
        use rich_ops, only: calc_rich_ex
        use vac_ops, only: calc_vac_res, print_output_vac
        use grid_utilities, only: trim_grid
        use x_ops, only: setup_modes, redistribute_output_x
#if ldebug
        use x_ops, only: print_debug_x_2
#endif

        character(*), parameter :: rout_name = 'run_driver_sol'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(grid_type), intent(in), target :: grid_x                           !< perturbation grid
        type(grid_type), intent(inout) :: grid_sol                              !< solution grid
        type(x_2_type), intent(in) :: x                                         !< integrated tensorial perturbation variables
        type(vac_type), intent(inout) :: vac                                    !< vacuum variables
        type(sol_type), intent(inout) :: sol                                    !< solution variables

        ! local variables
        type(grid_type) :: grid_sol_trim                                        ! trimmed solution grid
        type(grid_type) :: grid_x_rdst                                          ! redistributed perturbation grid
        type(grid_type) :: grid_x_sol                                           ! perturbation grid with solution normal part
        type(x_2_type) :: x_rdst                                                ! redistributed X
        type(x_2_type) :: x_sol                                                 ! interpolated X
        integer :: sol_limits(2)                                                ! min. and max. index of sol grid for this process
        integer :: rich_lvl_name                                                ! either the Richardson level or zero, to append to names
        real(dp), allocatable :: r_f_sol(:)                                     ! normal points in solution grid
        logical :: do_vac_ops                                                   ! whether specific calculations for vacuum are necessary
        character(len=max_str_ln) :: err_msg                                    ! error message

        ! initialize ierr
        ierr = 0

        ! set up solution grid if first level
        if (rich_lvl.eq.rich_restart_lvl) then
            ! Divide solution grid under group processes, calculating the limits
            ! and the normal coordinate.
            allocate(r_f_sol(n_r_sol))
            ierr = calc_norm_range('PB3D_sol',sol_limits=sol_limits,&
                &r_f_sol=r_f_sol)
            chckerr('')

            if (rich_lvl.eq.1) then
                call writo('Set up solution grid')
                call lvl_ud(1)

                call writo('Calculate the grid')
                call lvl_ud(1)
                ierr = setup_grid_sol(grid_eq,grid_x,grid_sol,r_f_sol,&
                    &sol_limits)
                chckerr('')
                call lvl_ud(-1)

                call writo('Write to output file')
                call lvl_ud(1)
                ierr = print_output_grid(grid_sol,'solution','sol',&
                    &remove_previous_arrs=(jump_to_sol.and.&
                    &(x_grid_style.eq.1 .or. x_grid_style.eq.3)))
                chckerr('')
                call lvl_ud(-1)

                ! set up modes
                ierr = setup_modes(mds_sol,grid_eq,grid_sol,plot_name='sol')
                chckerr('')

                call lvl_ud(-1)
            else
                ! restore solution grid and trim it
                ierr = reconstruct_pb3d_grid(grid_sol,'sol',&
```

```
                    &grid_limits=sol_limits)
                chckerr('')
                ierr = trim_grid(grid_sol,grid_sol_trim)
                chckerr('')

                ! set up modes
                ierr = setup_modes(mds_sol,grid_eq,grid_sol,plot_name='sol')
                chckerr('')

                ! reconstruct solution on trimmed grid
                ierr = reconstruct_pb3d_sol(mds_sol,grid_sol_trim,sol,'sol',&
                    &rich_lvl=rich_lvl-1)
                chckerr('')

                ! clean up
                call grid_sol_trim%dealloc()
            end if

            deallocate(r_f_sol)
        end if

        ! set up  whether Richardson level  has to be  appended to the  name and
        ! whether to do vacuum operations
        select case (eq_style)
            case (1)                                                    ! VMEC
                rich_lvl_name = rich_lvl                                ! append richardson level
                do_vac_ops = .true.
            case (2)                                                    ! HELENA
                rich_lvl_name = 0                                       ! do not append name
                if (rich_lvl.eq.rich_restart_lvl) then
                    do_vac_ops = .true.
                else
                    do_vac_ops = .false.
                end if
        end select

        if (do_vac_ops) then
            ! calculate vacuum
            ierr = calc_vac_res(mds_sol,vac)
            chckerr('')

            call writo('Write to output file')
            call lvl_ud(1)
            chckerr('')
            if (rank.eq.n_procs-1) then
                ierr = print_output_vac(vac,'vac',rich_lvl=rich_lvl_name)
                chckerr('')
            end if
            call lvl_ud(-1)
        end if

        ! initialize interpolated X
        ierr = grid_x_sol%init([1,n_alpha,grid_sol%n(3)],&
            &[grid_sol%i_min,grid_sol%i_max],grid_sol%divided)
        chckerr('')
        grid_x_sol%r_F = grid_sol%r_F
        grid_x_sol%r_E = grid_sol%r_E
        grid_x_sol%loc_r_F = grid_sol%loc_r_F
        grid_x_sol%loc_r_E = grid_sol%loc_r_E


        select case (x_grid_style)
            case (1,3)                                                  ! equilibrium or enriched
                ! user output
                call writo('Redistribute and interpolate the perturbation &
                    &variables to solution grid')
                call lvl_ud(1)

                ! redistribute grid and X variables
                ierr = redistribute_output_grid(grid_x,grid_x_rdst)
                chckerr('')
                ierr = redistribute_output_x(mds_x,grid_x,grid_x_rdst,x,x_rdst)
                chckerr('')

                ! initialize
                call x_sol%init(mds_sol,grid_x_sol,is_field_averaged=.true.)

                ! interpolate
                ierr = interp_v(mds_x,grid_x_rdst,x_rdst,mds_sol,grid_x_sol,&
                    &x_sol)
                chckerr('')

                ! clean up
                call x_rdst%dealloc()
                call grid_x_rdst%dealloc()

                call lvl_ud(-1)
```

```
            case (2)                                                        ! solution
                ! user output
                call writo('Copy the perturbation variables to solution grid')
                call lvl_ud(1)

                ! copy
                call x%copy(mds_sol,grid_x,x_sol)

                call lvl_ud(-1)
        end select

#if ldebug
        ! write integrated  field-aligned tensorial  perturbation quantities
        ! to output and plot
        if (debug_run_driver_sol) then
            ierr = print_debug_x_2(mds_sol,grid_x_sol,x_sol)
            chckerr('')
        end if
#endif

        ! clean up
        call grid_x_sol%dealloc()

        ! solve the system
        call writo('Solving the system')
        call lvl_ud(1)
        select case (ev_style)
            case(1)                                                         ! SLEPC solver for EV problem
                ! solve the system
                ierr = solve_ev_system_slepc(mds_sol,grid_sol,x_sol,vac,sol)
                chckerr('')
            case default
                err_msg = 'No EV solver style associated with '//&
                    &trim(i2str(ev_style))
                ierr = 1
                chckerr(err_msg)
        end select
        call lvl_ud(-1)
        call writo('System solved')

        ! write solution variables to output
        ierr = print_output_sol(grid_sol,sol,'sol',rich_lvl=rich_lvl,&
            &remove_previous_arrs=&
            &(jump_to_sol.and.(x_grid_style.eq.1 .or. x_grid_style.eq.3)))
        chckerr('')

        ! calculate Richardson extrapolation factors if necessary
        call calc_rich_ex(sol%val)

        ! clean up
        call x_sol%dealloc()
    end function run_driver_sol

    !> \public  Interpolate  tensorial  perturbation  quantities  in  the  third
    !! dimension.
    !!
    !! The input grid should not be divided, though the output grid can be.
    !!
    !! The procedure  considers all  possible mode  number combinations.  For \c
    !! X_style 2  (fast), each  secondary mode  only lives  in a  certain normal
    !! range  of the  plasma. Therefore,  each secondary  mode pair  also has  a
    !! limited normal range, given by the overlap of the ranges of the members.
    !!
    !! The interpolated mode number combinations  have a normal range that might
    !! slightly differ from the input ranges, in which case extrapolation can be
    !! done if the method allows for it.
    !!
    !! \note If the input grid is too  coarse, but the interpolated grid is not,
    !! it is in  theory possible that there are mode  numbers and therefore mode
    !! number pairs  that exist  in the  input grid,  though they  do so  in the
    !! interpolated  grid.  In  this  case,  they  can  not  be  calculated  and
    !! are  currently  set  to  zero. However,  the  current  implementation  of
    !! setup_modes()  produces an  error if  less than  the full  range of  mode
    !! numbers is accurately present.
    integer function interp_v(mds_i,grid_i,X_i,mds_o,grid_o,X_o) result(ierr)
        use x_utilities, only: is_necessary_x, trim_modes
        use x_vars, only: n_mod_x
        use num_vars, only: rank
        use eq_vars, only: max_flux_f
        use num_utilities, only: c
        use sol_utilities, only: interp_v_spline
!#if ldebug
        !use num_vars, only: ex_plot_style
!#endif

        character(*), parameter :: rout_name = 'interp_V'

```

```
        ! input / output
        type(modes_type), intent(in), target :: mds_i                          !< general modes variables for input
        type(grid_type), intent(in) :: grid_i                                  !< grid at which \c X_i is tabulated
        type(x_2_type), intent(in), target :: x_i                              !< tensorial perturbation variable on input grid
        type(modes_type), intent(in), target :: mds_o                          !< general modes variables for output
        type(grid_type), intent(in) :: grid_o                                  !< grid at which \c X_o is interpolated
        type(x_2_type), intent(inout), target :: x_o                           !< interpolated tensorial perturbation variable

        ! local variables
        integer :: kd                                                          ! counter
        integer :: k, m                                                        ! counters for mode numbers
        integer :: c_loc_i(2)                                                  ! local input c for symmetric and asymmetric variables
        integer :: c_loc_o(2)                                                  ! local output c for symmetric and asymmetric variables
        integer :: n_mod_tot                                                   ! total amount of modes
        integer :: kdl_i(2), kdl_o(2)                                          ! limits on normal index for a mode combination
        integer :: id_lim_i(2), id_lim_o(2)                                    ! limits on total modes
        integer :: ivs_stat(6)                                                 ! local stats for interp_V_spline
        integer, pointer :: sec_i_loc(:,:), sec_o_loc(:,:)                     ! pointers to secondary mode variables
        real(dp), pointer :: r_i_loc(:), r_o_loc(:)                            ! local r_i and r_o for a mode combination
        complex(dp), pointer :: v_i(:,:,:), v_o(:,:,:)                         ! pointers to input and output PV_i and KV_i
        logical :: calc_this(2)                                                ! whether this combination needs to be calculated
        logical :: extrap = .true.                                             ! whether extrapolation is used
        character(len=max_str_ln) :: err_msg                                   ! error message
#if ldebug
        integer :: km_id
        integer, allocatable :: norm_ext_i(:,:)                                ! normal extent for input quantity mode combinations
        real(dp), allocatable :: r_loc_tot(:,:,:)                              ! r_i_loc and r_o_loc for all combinations
        character(len=max_str_ln) :: ivs_stats_names(3)                        ! names used in interp_V_spline statistics
        !complex(dp), allocatable :: V_plot(:,:)                               ! for debug plotting of interpolated V
#endif

        ! initialize ierr
        ierr = 0

        ! test
        if (grid_i%n(2).ne.grid_o%n(2)) then
            ierr = 1
            err_msg = 'input and output grid are not compatible in the &
                &geodesic coordinate: '//trim(i2str(grid_i%n(2)))//' vs. '//&
                &trim(i2str(grid_o%n(2)))
            chckerr(err_msg)
        end if

        ! initialize to zero
        x_o%PV_0 = 0._dp
        x_o%PV_1 = 0._dp
        x_o%PV_2 = 0._dp
        x_o%KV_0 = 0._dp
        x_o%KV_1 = 0._dp
        x_o%KV_2 = 0._dp

        ! limit input modes to output modes
        ! (X grid should comprise sol grid)
        ierr = trim_modes(mds_i,mds_o,id_lim_i,id_lim_o)
        chckerr('')
        sec_i_loc => mds_i%sec(id_lim_i(1):id_lim_i(2),:)
        sec_o_loc => mds_o%sec(id_lim_o(1):id_lim_o(2),:)

        ! select all mode number combinations of interpolated grid
        n_mod_tot = size(sec_o_loc,1)
#if ldebug
        allocate(r_loc_tot(2,n_mod_tot**2,3))
        allocate(norm_ext_i(n_mod_tot,n_mod_tot))
        km_id = 0
        norm_ext_i = 0
#endif
        do m = 1,n_mod_tot
            do k = 1,n_mod_tot
#if ldebug
                ! test whether input and output mode numbers are consistent
                if (sec_i_loc(k,1).ne.sec_o_loc(k,1) .or. &
                    &sec_i_loc(m,1).ne.sec_o_loc(m,1)) then
                    ierr = 1
                    err_msg = 'For ('//trim(i2str(k))//','//trim(i2str(m))//&
                        &'), no consistency for modes in grid_i and grid_o'
                    chckerr(err_msg)
                end if
#endif

                ! set input and output grid normal limits for mode pair (k,m)
```

```
                kdl_i(1) = max(sec_i_loc(k,2),sec_i_loc(m,2))
                kdl_i(2) = min(sec_i_loc(k,3),sec_i_loc(m,3))
                kdl_o(1) = max(sec_o_loc(k,2),sec_o_loc(m,2))
                kdl_o(2) = min(sec_o_loc(k,3),sec_o_loc(m,3))

                ! limit to grid ranges
                kdl_i(1) = max(kdl_i(1),grid_i%i_min)
                kdl_i(2) = min(kdl_i(2),grid_i%i_max)
                kdl_o(1) = max(kdl_o(1),grid_o%i_min)
                kdl_o(2) = min(kdl_o(2),grid_o%i_max)

                ! limit output range to input range if not extrapolating
                if (.not.extrap) then
                    do kd = kdl_o(1),kdl_o(2)
                        if (grid_o%r_F(kd).ge.grid_i%r_F(kdl_i(1))) exit
                    end do
                    kdl_o(1) = kd

                    do kd = kdl_o(2),kdl_o(1),-1
                        if (grid_o%r_F(kd).le.grid_i%r_F(kdl_i(2))) exit
                    end do
                    kdl_o(2) = kd
                end if

                ! cycle if mode combination does not exist anywhere
                if (kdl_i(1).gt.kdl_i(2) .or. kdl_o(1).gt.kdl_o(2)) cycle

                ! convert limits to local
                kdl_i = kdl_i - grid_i%i_min + 1
                kdl_o = kdl_o - grid_o%i_min + 1

                ! get input ranges for this mode number
                r_i_loc => grid_i%loc_r_F(kdl_i(1):kdl_i(2))
                r_o_loc => grid_o%loc_r_F(kdl_o(1):kdl_o(2))

                ! check whether mode combination needs to be calculated
                calc_this(1) = is_necessary_x(.true.,&
                    &[sec_o_loc(k,1),sec_o_loc(m,1)])
                calc_this(2) = is_necessary_x(.false.,&
                    &[sec_o_loc(k,1),sec_o_loc(m,1)])

#if ldebug
                km_id = km_id + 1
                r_loc_tot(:,km_id,1) = [r_i_loc(1),r_i_loc(size(r_i_loc))]
                r_loc_tot(:,km_id,2) = [r_o_loc(1),r_o_loc(size(r_o_loc))]
                r_loc_tot(:,km_id,3) = km_id
                norm_ext_i(k,m) = size(r_i_loc)
                if (debug_interp_v) write(*,*) 'k, m', k, m, 'of', n_mod_tot, &
                        &'with calc_this = ', calc_this
#endif

                ! set up c_loc for input and output
                c_loc_i(1) = c([sec_i_loc(k,4),sec_i_loc(m,4)],.true.,n_mod_x)
                c_loc_i(2) = c([sec_i_loc(k,4),sec_i_loc(m,4)],.false.,n_mod_x)
                c_loc_o(1) = c([sec_o_loc(k,4),sec_o_loc(m,4)],.true.,n_mod_x)
                c_loc_o(2) = c([sec_o_loc(k,4),sec_o_loc(m,4)],.false.,n_mod_x)

                if (calc_this(1)) then
                    v_i => x_i%PV_0(:,:,kdl_i(1):kdl_i(2),c_loc_i(1))
                    v_o => x_o%PV_0(:,:,kdl_o(1):kdl_o(2),c_loc_o(1))
                    ierr = interp_v_spline(v_i,v_o,r_i_loc,r_o_loc,extrap,&
                        &ivs_stat(1))
                    chckerr('')

                    v_i => x_i%PV_2(:,:,kdl_i(1):kdl_i(2),c_loc_i(1))
                    v_o => x_o%PV_2(:,:,kdl_o(1):kdl_o(2),c_loc_o(1))
                    ierr = interp_v_spline(v_i,v_o,r_i_loc,r_o_loc,extrap,&
                        &ivs_stat(2))
                    chckerr('')

                    v_i => x_i%KV_0(:,:,kdl_i(1):kdl_i(2),c_loc_i(1))
                    v_o => x_o%KV_0(:,:,kdl_o(1):kdl_o(2),c_loc_o(1))
                    ierr = interp_v_spline(v_i,v_o,r_i_loc,r_o_loc,extrap,&
                        &ivs_stat(3))
                    chckerr('')

                    v_i => x_i%KV_2(:,:,kdl_i(1):kdl_i(2),c_loc_i(1))
                    v_o => x_o%KV_2(:,:,kdl_o(1):kdl_o(2),c_loc_o(1))
                    ierr = interp_v_spline(v_i,v_o,r_i_loc,r_o_loc,extrap,&
                        &ivs_stat(4))
                    chckerr('')

#if ldebug
                    if (debug_interp_v) then
                        !call writo('For [k,m] = ['//&
                            !&trim(i2str(sec_o_loc(k,1)))//','//&
                            !&trim(i2str(sec_o_loc(m,1)))//']:')
```

```
                        !call lvl_ud(1)
                        !call writo('input normal range: '//&
                            !&trim(i2str(kdl_i(1)))//'..'//trim(i2str(kdl_i(2))))
                        !call writo('output normal range: '//&
                            !&trim(i2str(kdl_o(1)))//'..'//trim(i2str(kdl_o(2))))
                        !allocate(V_plot(max(size(V_i,3),size(V_o,3)),4))
                        !V_plot(1:size(V_i,3),1) = V_i(1,1,:)
                        !V_plot(1:size(V_o,3),2) = V_o(1,1,:)
                        !V_plot(1:size(V_i,3),3) = r_i_loc/max_flux_F*2*pi
                        !V_plot(1:size(V_o,3),4) = r_o_loc/max_flux_F*2*pi
                        !V_plot(size(V_i,3)+1:size(V_plot,1),1) = &
                            !&V_plot(size(V_i,3),1)
                        !V_plot(size(V_o,3)+1:size(V_plot,1),2) = &
                            !&V_plot(size(V_o,3),2)
                        !V_plot(size(V_i,3)+1:size(V_plot,1),3) = &
                            !&V_plot(size(V_i,3),3)
                        !V_plot(size(V_o,3)+1:size(V_plot,1),4) = &
                            !&V_plot(size(V_o,3),4)
                        !call print_ex_2D(['V_i','V_o'],'RE_V',&
                            !&rp(V_plot(:,1:2)),x=rp(V_plot(:,3:4)))
                        !!call print_ex_2D(['V_i','V_o'],'IM_V',&
                            !!&ip(V_plot(:,1:2)),x=rp(V_plot(:,3:4)))
                        !deallocate(V_plot)
                        !if (ex_plot_style.eq.2) read(*,*)
                        !call lvl_ud(-1)
                    end if
#endif
                end if

                if (calc_this(2)) then
                    v_i => x_i%PV_1(:,:,kdl_i(1):kdl_i(2),c_loc_i(2))
                    v_o => x_o%PV_1(:,:,kdl_o(1):kdl_o(2),c_loc_o(2))
                    ierr = interp_v_spline(v_i,v_o,r_i_loc,r_o_loc,extrap,&
                        &ivs_stat(5))
                    chckerr('')

                    v_i => x_i%KV_1(:,:,kdl_i(1):kdl_i(2),c_loc_i(2))
                    v_o => x_o%KV_1(:,:,kdl_o(1):kdl_o(2),c_loc_o(2))
                    ierr = interp_v_spline(v_i,v_o,r_i_loc,r_o_loc,extrap,&
                        &ivs_stat(6))
                    chckerr('')
                end if
#if ldebug
                do kd = 1,size(ivs_stat)
                    !if (ivs_stat(kd).eq.1) call writo('Used direct copy for '//&
                        !&trim(i2str(kd))//' of mode combination ('//&
                        !&trim(i2str(k))//', '//trim(i2str(m))//') at ψ = '//&
                        !&trim(r2strt(r_loc_tot(1,km_id,1)/max_flux_F*2*pi)))
                    ivs_stats(ivs_stat(kd)) = ivs_stats(ivs_stat(kd)) + 1
                end do
#endif
            end do
        end do

#if ldebug
        call writo('interp_V_spline statistics:')
        call lvl_ud(1)
        ivs_stats_names(1) = 'direct copy'
        ivs_stats_names(2) = 'linear interpolation'
        ivs_stats_names(3) = 'spline interpolation'
        do kd = 1,3
            call writo(trim(ivs_stats_names(kd))//' was performed '//&
                &trim(i2str(ivs_stats(kd)))//' times: '//&
                &trim(r2strt(100._dp*ivs_stats(kd)/sum(ivs_stats)))//'%')
        end do
        if (ivs_stats(1).gt.0) then
            call writo('It is possible that more secondary modes and/or a &
                &lower max_njq_change are needed.',warning=.true.)
            call writo('If unphysical modes appear, consider changing these &
                &variables.',warning=.true.)
        end if
        call lvl_ud(-1)
        call print_ex_2d([''],'r_i_loc_'//trim(i2str(rank)),&
            &r_loc_tot(:,1:km_id,1)/max_flux_f*2*pi,x=r_loc_tot(:,1:km_id,3),&
            &draw=.false.)
        call draw_ex([''],'r_i_loc_'//trim(i2str(rank)),km_id,1,.false.,&
            &ex_plot_style=1)
        call print_ex_2d([''],'r_o_loc_'//trim(i2str(rank)),&
            &r_loc_tot(:,1:km_id,2)/max_flux_f*2*pi,x=r_loc_tot(:,1:km_id,3),&
            &draw=.false.)
        call draw_ex([''],'r_o_loc_'//trim(i2str(rank)),km_id,1,.false.,&
            &ex_plot_style=1)
        call plot_hdf5('normal extent','norm_ext_i_'//trim(i2str(rank)),1._dp*&
            &reshape(norm_ext_i*1._dp,[n_mod_tot,n_mod_tot,1]))
#endif

        ! clean up
```

```
        nullify(r_i_loc,r_o_loc)
        nullify(sec_i_loc,sec_o_loc)
        nullify(v_i,v_o)
    end function interp_v
end module driver_sol
```

D.14 /Users/toonweyens/Code/PB3D/Modules/driver_X.f90 File Reference

Modules

- module driver_x

  Driver of the perturbation part of PB3D.

Functions/Subroutines

- integer function, public driver_x::run_driver_x (grid_eq, grid_eq_b, grid_x, grid_x_b, eq_1, eq_2, x_1, x_2)

  Main driver of PB3D perturbation part.

D.15 driver_X.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Driver of the perturbation part of PB3D.
!-----------------------------------------------------------------------------!
module driver_x
#include <PB3D_macros.h>
#include <wrappers.h>
    use str_utilities
    use output_ops
    use messages
    use num_vars, only: dp, pi, max_str_ln
    use grid_vars, only: grid_type
    use eq_vars, only: eq_1_type, eq_2_type
    use x_vars, only: x_1_type, x_2_type, modes_type, &
        &mds_x

    implicit none
    private
    public run_driver_x

    ! global variables
    integer :: X_limits(2)                                          !< min. and max. index of X grid for this process
    real(dp), allocatable :: r_F_X(:)                               !< normal points in perturbation grid
    character(len=8) :: flux_name(2)                                !< name of flux variable
    character(len=1) :: mode_name(2)                                !< name of modes
    integer :: rich_lvl_name                                        !< either the Richardson level or zero
#if ldebug
    logical :: debug_run_driver_X_1 = .false.                       !< debug information for run_driver_X_1 \ldebug
    logical :: debug_run_driver_X_2 = .false.                       !< debug information for run_driver_X_2 \ldebug
    logical :: plot_info= .false.                                   !< plot information for comparison with HELENA \ldebug
#endif

contains
    !> Main driver of PB3D perturbation part.
    !!
    !!  - sets up:
    !!      * [0] \c grid_X (for HELENA, only first Richardson level)
    !!      * [0] \c grid_X_B (for VMEC, equal to grid_X_out)
```

```
    !!      * [1] \c X_1
    !!      * [2] \c X_2
    !!  - writes to HDF5:
    !!      * [0] \c grid_X (for HELENA, only first Richardson level)
    !!      * [0] \c grid_X_B (for VMEC, equal to grid_X)
    !!      * [1] \c X_1 (only for HELENA, only first Richardson level)
    !!      * [2] \c X_2
    !!  - deallocates:
    !!      * \c X_1 before setting up
    !!      * \c grid_X before setting up
    !!      * \c grid_X_B before setting up
    !!
    !!  ([x] indicates driver x)
    !!
    !! \note  \c eq_2  needs  to be  intent(inout) because  interp_HEL_on_grid()
    !! requires  this for  generality.  The  variable is  not  modified in  this
    !! driver, though.
    !!
    !! \return ierr
    integer function run_driver_x(grid_eq,grid_eq_B,grid_X,grid_X_B,eq_1,eq_2,&
        &X_1,X_2) result(ierr)
        use num_vars, only: use_pol_flux_f, eq_style, plot_resonance, &
            &x_style, rich_restart_lvl, jump_to_sol, eq_job_nr, x_grid_style
        use rich_vars, only: n_par_x, rich_lvl
        use pb3d_ops, only: reconstruct_pb3d_grid, reconstruct_pb3d_x_1, &
            &reconstruct_pb3d_x_2
        use x_vars, only: min_sec_x, max_sec_x, prim_x, min_r_sol, max_r_sol, &
            &n_mod_x
        use grid_ops, only: calc_norm_range
        use grid_vars, only: min_par_x, max_par_x, n_r_sol
        use x_ops, only: calc_x, check_x_modes, resonance_plot, &
            &init_modes, setup_modes
        use files_utilities, only: delete_file
        use grid_utilities, only: trim_grid
        use mpi_utilities, only: get_ser_var

        character(*), parameter :: rout_name = 'run_driver_X'

        ! input / output
        type(grid_type), intent(in), target :: grid_eq                          !< equilibrium grid (should be in but needs inout for interp_HEL_on_grid)
        type(grid_type), intent(in), pointer :: grid_eq_b                       !< field-aligned equilibrium grid (should be in but needs inout for interp_HEL_on_grid)
        type(grid_type), intent(inout), target :: grid_x                        !< perturbation grid
        type(grid_type), intent(inout), pointer :: grid_x_b                     !< field-aligned perturbation grid
        type(eq_1_type), intent(in), target :: eq_1                             !< flux equilibrium variables
        type(eq_2_type), intent(inout), target :: eq_2                          !< metric equilibrium variables (should be in but needs inout for interp_HEL_on_grid)
        type(x_1_type), intent(inout) :: x_1                                    !< vectorial perturbation variables
        type(x_2_type), intent(inout) :: x_2                                    !< tensorial perturbation variables

        ! local variables
        type(grid_type) :: grid_eq_trim                                         ! trimmed equilibrium grid
        type(eq_2_type), pointer :: eq_2_b                                      ! field-aligned metric equilibrium variables
        integer :: eq_limits(2)                                                 ! min. and max. index of eq. grid for this process
        integer :: norm_id(2)                                                   ! untrimmed normal indices for trimmed grids
        real(dp), pointer :: r_f_eq(:)                                          ! pointer to r_F of grid_eq
        real(dp), pointer :: jq(:)                                              ! q_saf (pol. flux) or rot_t (tor. flux) in Flux variables
        real(dp), allocatable :: jq_ser(:)                                      ! serial jq

        ! initialize ierr
        ierr = 0

        ! set up whether Richardson level has to be appended to the name
        select case (eq_style)
            case (1)                                                            ! VMEC
                rich_lvl_name = rich_lvl                                        ! append richardson level
            case (2)                                                            ! HELENA
                rich_lvl_name = 0                                               ! do not append
        end select

        ! Divide perturbation grid under group processes, calculating the limits
        ! and the normal coordinate.
        if (rich_lvl.eq.rich_restart_lvl .and. eq_job_nr.eq.1) then
            ! initialize helper variables
            ierr = trim_grid(grid_eq,grid_eq_trim,norm_id)
            chckerr('')
            if (use_pol_flux_f) then
                jq => eq_1%q_saf_FD(:,0)
            else
                jq => eq_1%rot_t_FD(:,0)
            end if
            ierr = get_ser_var(jq(norm_id(1):norm_id(2)),jq_ser,scatter=.true.)
            chckerr('')
            call grid_eq_trim%dealloc()
```

```
            eq_limits = [grid_eq%i_min,grid_eq%i_max]

            ! calculate normal range
            call writo('Set up perturbation normal range')
            call lvl_ud(1)
            r_f_eq => grid_eq%r_F                                          ! needed for compiler bug on Intel 12.0.2
            ierr = calc_norm_range('PB3D_X',eq_limits=eq_limits,&
                &x_limits=x_limits,r_f_eq=r_f_eq,r_f_x=r_f_x,jq=jq_ser)
            chckerr('')
            nullify(r_f_eq)
            call lvl_ud(-1)

            ! clean up
            nullify(jq)
        end if

        ! jump to solution if requested
        if (rich_lvl.eq.rich_restart_lvl .and. jump_to_sol) then
            call writo('Prepare to jump to solution')
            call lvl_ud(1)

            ! grid_X
            ierr = reconstruct_pb3d_grid(grid_x,'X',rich_lvl=rich_lvl_name,&
                &grid_limits=x_limits)
            chckerr('')

            ! initialize modes and set up
            ierr = init_modes(grid_eq,eq_1)
            chckerr('')
            ierr = setup_modes(mds_x,grid_eq,grid_x,plot_name='X')
            chckerr('')

            ! tests
            ierr = check_x_modes(grid_eq,eq_1)
            chckerr('')

            ! X_1
            if (eq_style.eq.2) then                                        ! HELENA
                ierr = reconstruct_pb3d_x_1(mds_x,grid_x,x_1,'X_1')
                chckerr('')
            end if

            ! X_2
            ierr = reconstruct_pb3d_x_2(mds_x,grid_x,x_2,'X_2_int',&
                &rich_lvl=rich_lvl,is_field_averaged=.true.)
            chckerr('')

            call lvl_ud(-1)
            call writo('Skipping rest to jump to solution')
            return
        end if

        ! user output
        call writo('Perturbations are analyzed')
        call lvl_ud(1)

        call writo('for '//trim(i2str(size(r_f_x)))//&
            &' values on normal range '//trim(r2strt(min_r_sol))//'..'//&
            &trim(r2strt(max_r_sol)))
        select case (x_grid_style)
            case (1,3)                                                     ! equilibrium or enriched
                call lvl_ud(1)
                call writo('interpolation to the solution grid with '//&
                    &trim(i2str(n_r_sol))//' points will happen in the &
                    &solution driver')
                call lvl_ud(-1)
            case (2)                                                       ! solution
                ! do nothing
        end select
        call writo('for '//trim(i2str(n_par_x))//' values on parallel &
            &range '//trim(r2strt(min_par_x*pi))//'..'//&
            &trim(r2strt(max_par_x*pi)))

        ! set flux and mode names
        if (use_pol_flux_f) then
            flux_name = ['poloidal','toroidal']
            mode_name = ['m','n']
        else
            flux_name = ['toroidal','poloidal']
            mode_name = ['n','m']
        end if

        ! user output
        call writo('with '//flux_name(2)//' mode number '//mode_name(2)//&
            &' = '//trim(i2str(prim_x)))
        select case(x_style)
            case (1)                                                       ! prescribed
```

```
                call writo('and '//flux_name(1)//' mode number '//&
                    &mode_name(1)//' = '//trim(i2str(min_sec_x))//'..'//&
                    &trim(i2str(max_sec_x)))
            case (2)                                                    ! fast
                call writo('and the '//trim(i2str(n_mod_x))//&
                    &' '//flux_name(1)//' modes '//mode_name(1)//&
                    &' that resonate most')
        end select

        call lvl_ud(-1)

        ! Run grid part of driver
        ierr = run_driver_x_0(grid_eq,grid_eq_b,grid_x,grid_x_b,eq_1,&
            &eq_2,eq_2_b)
        chckerr('')

        ! setup nm and check the X modes if restart Richardson level
        if (rich_lvl.eq.rich_restart_lvl .and. eq_job_nr.eq.1) then
            ! initialize modes and set up
            ierr = init_modes(grid_eq,eq_1)
            chckerr('')
            ierr = setup_modes(mds_x,grid_eq,grid_x,plot_name='X')
            chckerr('')

            ! tests
            ierr = check_x_modes(grid_eq,eq_1)
            chckerr('')
        end if

        ! plot resonances if requested
        if (plot_resonance .and. rich_lvl.eq.1 .and. eq_job_nr.eq.1) then
            ierr = resonance_plot(mds_x,grid_eq,eq_1)
            chckerr('')
        else
            call writo('Resonance plot not requested')
        end if

        ! Run vectorial part of driver
        ierr = run_driver_x_1(grid_eq,grid_x,eq_1,eq_2,x_1)
        chckerr('')

        ! Run Tensorial part of driver
        ierr = run_driver_x_2(grid_eq_b,grid_x,grid_x_b,eq_1,eq_2_b,x_1,x_2)
        chckerr('')

        ! clean up
        call writo('Clean up')
        call lvl_ud(1)
        if (eq_style.eq.2) then
            call eq_2_b%dealloc()
            deallocate(eq_2_b)
        end if
        nullify(eq_2_b)
        call lvl_ud(-1)
    end function run_driver_x

    !> part  0 of  driver_x:  perturbation  grid as  well  as reconstruction  of
    !! variables.
    integer function run_driver_x_0(grid_eq,grid_eq_B,grid_X,grid_X_B,eq_1,&
        &eq_2,eq_2_B) result(ierr)
        use num_vars, only: eq_style, eq_job_nr, rich_restart_lvl, eq_jobs_lims
        use grid_ops, only: setup_grid_x, print_output_grid
        use rich_vars, only: rich_lvl
        use helena_ops, only: interp_hel_on_grid

        character(*), parameter :: rout_name = 'run_driver_X_0'

        ! input / output
        type(grid_type), intent(in), target :: grid_eq                  !< equilibrium grid
        type(grid_type), intent(in), pointer :: grid_eq_b               !< field-aligned equilibrium grid
        type(grid_type), intent(inout), target :: grid_x                !< perturbation grid
        type(grid_type), intent(inout), pointer :: grid_x_b             !< field-aligned perturbation grid
        type(eq_1_type), intent(in) :: eq_1                             !< flux equilibrium variables
        type(eq_2_type), intent(inout), target :: eq_2                  !< metric equilibrium variables
        type(eq_2_type), intent(inout), pointer :: eq_2_b               !< field-aligned metric equilibrium
variables

        ! local variables
        logical :: do_x_2_ops                                           ! whether specific calculations for X_2 are
necessary
        integer :: rich_lvl_name                                        ! either the Richardson level or zero, to
append to names

        ! initialize ierr
        ierr = 0

        ! user output
```

```
        call writo('Setting up perturbation grid')
        call lvl_ud(1)

        ! decide whether to do certain  perturbation calculations and whether to
        ! append Richardson level to name
        select case (eq_style)
            case (1)                                                        ! VMEC
                do_x_2_ops = .true.
                rich_lvl_name = rich_lvl                                    ! append richardson level
            case (2)                                                        ! HELENA
                if (rich_lvl.eq.rich_restart_lvl .and. eq_job_nr.eq.1) then
                    do_x_2_ops = .true.
                else
                    do_x_2_ops = .false.
                end if
                rich_lvl_name = 0                                           ! do not append name
        end select

        call writo('Reconstruct equilibrium variables')
        call lvl_ud(1)
        select case (eq_style)
            case (1)                                                        ! VMEC
                ! point: grid is already field-aligned
                eq_2_b => eq_2
            case (2)                                                        ! HELENA
                ! allocate field-aligned equilibrium variables
                allocate(eq_2_b)

                ! interpolate field-aligned metric equilibrium variables
                call eq_2_b%init(grid_eq_b,setup_e=.false.,setup_f=.true.)
                ierr = interp_hel_on_grid(grid_eq,grid_eq_b,eq_2=eq_2,&
                    &eq_2_out=eq_2_b,eq_1=eq_1,grid_name='field-aligned &
                    &equilibrium grid')
                chckerr('')
        end select
        call lvl_ud(-1)

        ! create and setup perturbation grid with division limits
        if (do_x_2_ops) then
            call writo('Determine the perturbation grid')
            call lvl_ud(1)
            if (associated(grid_x%r_F)) call grid_x%dealloc()               ! deallocate if necessary
            ierr = setup_grid_x(grid_eq,grid_x,r_f_x,x_limits)
            chckerr('')
            call lvl_ud(-1)

            ! print output
            ierr = print_output_grid(grid_x,'perturbation','X',&
                &rich_lvl=rich_lvl_name,par_div=size(eq_jobs_lims,2).gt.1)
            chckerr('')
        end if

        ! grid_X_B
        select case (eq_style)
            case (1)                                                        ! VMEC
                ! do nothing: grid is already field-aligned
                grid_x_b => grid_x
            case (2)                                                        ! HELENA
                ! set up field-aligned perturbation grid
                if (associated(grid_x_b)) then
                    call grid_x_b%dealloc()
                    deallocate(grid_x_b)
                    nullify(grid_x_b)
                end if
                allocate(grid_x_b)
                ierr = setup_grid_x(grid_eq_b,grid_x_b,r_f_x,x_limits)
                chckerr('')

                ! print field-aligned output
                ierr = print_output_grid(grid_x_b,'field-aligned &
                    &perturbation','X_B',rich_lvl=rich_lvl,par_div=.true.)
                chckerr('')
        end select

        call lvl_ud(-1)
        call writo('Perturbation grid set up')
    end function run_driver_x_0

    !> Part 1 of driver_x: Vectorial jobs.
    !!
    !! \note  Everything is  done here  in the  original grids,  not necessarily
    !! field-aligned.
    !!
    !! \return ierr
    integer function run_driver_x_1(grid_eq,grid_X,eq_1,eq_2,X_1) result(ierr)
        use num_vars, only: eq_style, u_style, eq_job_nr, rich_restart_lvl
        use x_ops, only: calc_x, print_output_x
```

```
        use rich_vars, only: rich_lvl
#if ldebug
        use x_ops, only: print_debug_x_1
#endif

        character(*), parameter :: rout_name = 'run_driver_X_1'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(grid_type), intent(in) :: grid_x                                   !< perturbation grid
        type(eq_1_type), intent(in) :: eq_1                                     !< flux equilibrium variables
        type(eq_2_type), intent(in) :: eq_2                                     !< metric equilibrium variables
        type(x_1_type), intent(inout) :: x_1                                    !< vectorial X variables

        ! local variables
        logical :: do_x_1_ops                                                   ! whether specific calculations for X_1 are necessary

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Calculating U up to order '//trim(i2str(u_style)))

        ! decide whether to do certain  perturbation calculations and whether to
        ! append Richardson level to name
        select case (eq_style)
            case (1)                                                            ! VMEC
                do_x_1_ops = .true.
            case (2)                                                            ! HELENA
                if (rich_lvl.eq.rich_restart_lvl .and. eq_job_nr.eq.1) then
                    do_x_1_ops = .true.
                else
                    do_x_1_ops = .false.
                end if
        end select

        if (do_x_1_ops) then
            ! possibly deallocate
            if (allocated(x_1%U_0)) call x_1%dealloc()

            ! calc variables
            ierr = calc_x(mds_x,grid_eq,grid_x,eq_1,eq_2,x_1)
            chckerr('')
        else
            ! For HELENA, everything has been done
            return
        end if

        ! write vectorial perturbation variables to output if HELENA
        ! Note: There should be only 1 parallel job
        if (eq_style.eq.2) then                                                 ! only if first Richardson level
            ierr = print_output_x(grid_x,x_1,'X_1',par_div=.false.)
            chckerr('')
        end if

#if ldebug
        ! write vectorial perturbation quantities to output
        if (debug_run_driver_x_1) then
            ierr = print_debug_x_1(mds_x,grid_x,x_1)
            chckerr('')
        end if

        if (plot_info) then
            ! plot information for comparison between VMEC and HELENA
            call plot_info_for_vmec_hel_comparison(grid_x,x_1)
        end if
#endif
#if ldebug
    contains
        !> \private
        subroutine plot_info_for_vmec_hel_comparison(grid_X,X)
            use input_utilities, only: pause_prog, get_int, get_log
            use num_vars, only: use_pol_flux_f

            ! input / output
            type(grid_type), intent(in) :: grid_x
            type(x_1_type), intent(in) :: x

            ! local variables
            real(dp), allocatable :: x_plot(:,:,:), y_plot(:,:,:), z_plot(:,:,:)
            integer :: id, ld
            logical :: not_ready

            call writo('Plotting information for comparison between VMEC and &
                &HELENA',alert=.true.)
            not_ready = .true.
```

```
            do while (not_ready)
                call writo('Plots for which mode number?')
                ld =  get_int(lim_lo=1,lim_hi=x%n_mod)

                ! x, y, z
                allocate(x_plot(grid_x%n(1),grid_x%n(2),grid_x%loc_n_r))
                allocate(y_plot(grid_x%n(1),grid_x%n(2),grid_x%loc_n_r))
                allocate(z_plot(grid_x%n(1),grid_x%n(2),grid_x%loc_n_r))
                if (use_pol_flux_f) then
                    x_plot = grid_x%theta_F
                else
                    x_plot = grid_x%zeta_F
                end if
                y_plot = 0._dp
                do id = 1,grid_x%loc_n_r
                    z_plot(:,:,id) = grid_x%loc_r_F(id)/maxval(grid_x%r_F)
                end do
                call plot_hdf5('x_plot','x_plot',x_plot)
                call plot_hdf5('y_plot','y_plot',y_plot)
                call plot_hdf5('z_plot','z_plot',z_plot)

                ! U_0 and U_1
                call plot_hdf5('RE_U_0','RE_U_0',rp(x%U_0(:,:,:,ld)),&
                    &x=x_plot,y=y_plot,z=z_plot)
                call plot_hdf5('IM_U_0','IM_U_0',ip(x%U_0(:,:,:,ld)),&
                    &x=x_plot,y=y_plot,z=z_plot)
                call plot_hdf5('RE_U_1','RE_U_1',rp(x%U_1(:,:,:,ld)),&
                    &x=x_plot,y=y_plot,z=z_plot)
                call plot_hdf5('IM_U_1','IM_U_1',ip(x%U_1(:,:,:,ld)),&
                    &x=x_plot,y=y_plot,z=z_plot)

                ! DU_0 and DU_1
                call plot_hdf5('RE_DU_0','RE_DU_0',rp(x%DU_0(:,:,:,ld)),&
                    &x=x_plot,y=y_plot,z=z_plot)
                call plot_hdf5('IM_DU_0','IM_DU_0',ip(x%DU_0(:,:,:,ld)),&
                    &x=x_plot,y=y_plot,z=z_plot)
                call plot_hdf5('RE_DU_1','RE_DU_1',rp(x%DU_1(:,:,:,ld)),&
                    &x=x_plot,y=y_plot,z=z_plot)
                call plot_hdf5('IM_DU_1','IM_DU_1',ip(x%DU_1(:,:,:,ld)),&
                    &x=x_plot,y=y_plot,z=z_plot)

                ! clean up
                deallocate(x_plot,y_plot,z_plot)

                call writo('Want to redo the plotting?')
                not_ready = get_log(.true.)
            end do

            call writo('Done, paused',alert=.true.)
            call pause_prog()
        end subroutine plot_info_for_vmec_hel_comparison
#endif
    end function run_driver_x_1

    !> Part 2 of driver_X: Tensorial jobs.
    !!
    !! \note  Everything  is  done  in the  field-aligned  grids,  where  HELENA
    !! vectorial perturbation variables are first interpolated.
    !!
    !! \return ierr
    integer function run_driver_x_2(grid_eq_B,grid_X,grid_X_B,eq_1,eq_2_B,X_1,&
        &X_2_int) result(ierr)
        use pb3d_ops, only: reconstruct_pb3d_x_2
        use num_vars, only: x_job_nr, x_jobs_lims, eq_style, eq_job_nr, &
            &eq_jobs_lims, magn_int_style, no_output, rich_restart_lvl
        use rich_vars, only: rich_lvl
        use x_ops, only: calc_x, print_output_x, calc_magn_ints, divide_x_jobs
        use helena_ops, only: interp_hel_on_grid
        use x_utilities, only: do_x
#if ldebug
        use x_ops, only: print_debug_x_2
#endif

        character(*), parameter :: rout_name = 'run_driver_X_2'

        ! input / output
        type(grid_type), intent(in), pointer :: grid_eq_b                      !< field-aligned equilibrium grid
        type(grid_type), intent(in), target :: grid_x                           !< perturbation grid
        type(grid_type), intent(in), pointer :: grid_x_b                        !< field-aligned perturbation grid
        type(eq_1_type), intent(in) :: eq_1                                     !< flux equilibrium variables
        type(eq_2_type), intent(in), pointer :: eq_2_b                          !< field-aligned metric equilibrium
    variables
        type(x_1_type), intent(in) :: x_1                                       !< vectorial X variables
        type(x_2_type), intent(inout) :: x_2_int                                !< tensorial X variables

        ! local variables
        logical :: no_output_loc                                                ! local copy of no_output
```

```
        type(x_1_type) :: x_1_loc(2)                                        ! local X_1
        type(x_2_type) :: x_2                                               ! tensorial X variables
        integer :: id                                                       ! counter
        integer :: var_size_without_par                                     ! size of array for jobs division
        integer :: prev_style                                               ! style to treat prev_X
        integer :: lims_loc(2,2)                                            ! local X_jobs_lims

        ! initialize ierr
        ierr = 0

        call writo('Setting up Tensorial perturbation jobs')
        call lvl_ud(1)

        ! divide perturbation jobs, tensor phase
        var_size_without_par = ceiling(product(grid_x_b%n(1:3))*1._dp)
        ierr = divide_x_jobs(var_size_without_par)
        chckerr('')

        ! if first equilibrium job of  first Richardson level, set up integrated
        ! tensorial perturbation variables
        if (rich_lvl.eq.rich_restart_lvl .and. eq_job_nr.eq.1) then
            if (rich_lvl.eq.1) then
                call x_2_int%init(mds_x,grid_x_b,is_field_averaged=.true.)
                x_2_int%PV_0 = 0._dp
                x_2_int%PV_1 = 0._dp
                x_2_int%PV_2 = 0._dp
                x_2_int%KV_0 = 0._dp
                x_2_int%KV_1 = 0._dp
                x_2_int%KV_2 = 0._dp
            else
                ierr = reconstruct_pb3d_x_2(mds_x,grid_x,x_2_int,'X_2_int',&
                    &rich_lvl=rich_lvl-1,is_field_averaged=.true.)
                chckerr('')
            end if
        end if

        ! user output
        call writo('The interpolation method used for the magnetic integrals:')
        call lvl_ud(1)
        select case (magn_int_style)
            case (1)
                call writo('magnetic interpolation style: 1 (trapezoidal rule)')
            case (2)
                call writo('magnetic interpolation style: 2 (Simpson 3/8 rule)')
        end select
        call lvl_ud(-1)

        call lvl_ud(-1)
        call writo('Tensorial perturbation jobs set up')

        ! main loop over tensorial jobs
        x_job_nr = 0
        call writo('Starting tensorial perturbation jobs')
        call lvl_ud(1)
        x_jobs: do while(do_x())
            ! user output
            call print_info_x_2()
            call lvl_ud(1)
            no_output_loc = no_output
            no_output = .true.

            ! set local X_jobs limits
            lims_loc = reshape(x_jobs_lims(:,x_job_nr),[2,2])

            ! set local X_1
            do id = 1,2
                ! create X_1_loc
                call x_1_loc(id)%init(mds_x,grid_x,lim_sec_x=lims_loc(:,id))

                ! fill X_1_loc
                x_1_loc(id)%U_0 = x_1%U_0(:,:,:,lims_loc(1,id):lims_loc(2,id))
                x_1_loc(id)%U_1 = x_1%U_1(:,:,:,lims_loc(1,id):lims_loc(2,id))
                x_1_loc(id)%DU_0 = x_1%DU_0(:,:,:,lims_loc(1,id):lims_loc(2,id))
                x_1_loc(id)%DU_1 = x_1%DU_1(:,:,:,lims_loc(1,id):lims_loc(2,id))

                ! possibly interpolate
                select case (eq_style)
                    case (1)                                                ! VMEC
                        ! do nothing
                    case (2)                                                ! HELENA
                        ierr = interp_hel_on_grid(grid_x,grid_x_b,&
                            &x_1=x_1_loc(id),grid_name=&
                            &'field-aligned perturbation grid')
                        chckerr('')
                end select
            end do

```

```
            ! calculate X variables, tensor phase
            ierr = calc_x(mds_x,grid_eq_b,grid_x_b,eq_1,eq_2_b,&
                &x_1_loc(1),x_1_loc(2),x_2,lim_sec_x=lims_loc)
            chckerr('')

            ! integrate magnetic  integrals of tensorial  perturbation variables
            ! over  field-aligned  grid.  If   not  first  Richardson  level  or
            ! equilibrium job, previous integrated X_2 is used:
            !   - rich_lvl = 1, eq_job_nr = 1: just calculate integral
            !   - rich_lvl = 1, eq_job_nr > 1: sum to previous integral
            !   - rich_lvl > 1, eq_job_nr = 1: sum to previous integral / 2
            !   - rich_lvl > 1, eq_job_nr > 1: sum to previous integral
            if (rich_lvl.eq.1) then
                if (eq_job_nr.eq.1) then
                    prev_style = 0                                          ! don't add
                else
                    prev_style = 1                                          ! add
                end if
            else
                if (eq_job_nr.eq.1) then
                    prev_style = 2                                          ! divide by 2 and add, change indices
                else
                    prev_style = 3                                          ! add, change indices
                end if
            end if
            ierr = calc_magn_ints(grid_eq_b,grid_x_b,eq_2_b,x_2,x_2_int,&
                &prev_style,lim_sec_x=lims_loc)
            chckerr('')

            ! clean up
            do id = 1,2
                call x_1_loc(id)%dealloc()
            end do
            call x_2%dealloc()

            ! user output
            no_output = no_output_loc
            call lvl_ud(-1)
        end do x_jobs

#if ldebug
        ! write  integrated field-aligned  tensorial perturbation  quantities to
        ! output and plot
        if (debug_run_driver_x_2) then
            ierr = print_debug_x_2(mds_x,grid_x,x_2_int)
            chckerr('')
        end if
#endif

        call lvl_ud(-1)
        call writo('Tensorial perturbation jobs finished')

        ! write field-averaged tensorial perturbation variables to output
        if (eq_job_nr.eq.size(eq_jobs_lims,2)) then
            ierr = print_output_x(grid_x_b,x_2_int,'X_2_int',rich_lvl=rich_lvl,&
                &is_field_averaged=.true.)
            chckerr('')
        end if
    end function run_driver_x_2

    !> Prints information for tensorial perturbation job.
    subroutine print_info_x_2()
        use num_vars, only: x_job_nr, x_jobs_lims

        ! user output
        call writo('Job '//trim(i2str(x_job_nr))//'/'//&
            &trim(i2str(size(x_jobs_lims,2)))//': mode numbers ('//&
            &trim(i2str(x_jobs_lims(1,x_job_nr)))//'..'//&
            &trim(i2str(x_jobs_lims(2,x_job_nr)))//')x('//&
            &trim(i2str(x_jobs_lims(3,x_job_nr)))//'..'//&
            &trim(i2str(x_jobs_lims(4,x_job_nr)))//')')
    end subroutine print_info_x_2
end module driver_x
```

D.16 /Users/toonweyens/Code/PB3D/Modules/dtorh.f90 File Reference

Modules

- module dtorh

Calculation of toroidal functions $P^m_{n-1/2}(z)$ and $Q^m_{n-1/2}(z)$.

Functions/Subroutines

- integer function, public dtorh::dtorh1 (z, m, nmax, pl, ql, newn, mode, ipre)

  Calculates toroidal harmonics of a fixed order m and degrees up to nmax.

D.17 dtorh.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Calculation of toroidal functions \f$P_{n-1/2}^m \left(z\right)\f$ and
!! \f$Q_{n-1/2}^m \left(z\right)\f$.
!-----------------------------------------------------------------------------!
!> \note Copied and adapted from the DTORH1 routine by Segura and Gil
!! \cite Segura2000 .
!-----------------------------------------------------------------------------!
module dtorh
#include <PB3D_macros.h>
    use num_vars, only: dp, pi, max_str_ln
    use messages
    use str_utilities

    implicit none
    private
    public dtorh1

   character(len=max_str_ln) :: err_msg                                    !< error message

contains
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !  INPUT :                                                              !
    !    Z        ARGUMENT OF THE FUNCTIONS                                 !
    !    M        ORDER OF THE FUNCTIONS                                    !
    !    NMAX     MAXIMUM DEGREE OF THE FUNCTIONS :                         !
    !             WE GET  FUNCTIONS OF ALL THE ORDERS BELOW                 !
    !             MIN(NEWN,NMAX). NEWN IS DEFINED BELOW .                   !
    !  OUTPUT :                                                             !
    !   *IF MODE IS EQUAL TO 0:                                             !
    !    PL(N)                                                              !
    !             THESE VALUES ARE KEPT IN AN ARRAY                         !
    !    QL(N)                                                              !
    !             THESE VALUES ARE KEPT IN AN ARRAY                         !
    !                                                                        !
    !    NEWN     MAXIMUM ORDER OF FUNCTIONS CALCULATED WHEN                !
    !             PL (NMAX+1) IS LARGER THAN 1/TINY                         !
    !             (OVERFLOW LIMIT = 1/TINY, TINY IS DEFINED BELOW)          !
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !> Calculates toroidal harmonics of a fixed order \c m and degrees
    !! up to \c nmax.
    !!
    !! Optionally, the \c mode can be specified to be different from 0 [default]
    !!  - if \c mode=1:
    !!      - The set of functions evaluated is:
    !!        \f[\frac{P_{n-1/2}^m \left(z\right)}{\Gamma(m+1/2)} \ ,\quad
    !!          \frac{Q_{n-1/2}^m \left(z\right)}{\Gamma(m+1/2)},\f]
    !!        which are respectively stored in the arrays \c pl(n), \c ql(n).
    !!      - newn refers to this new set of functions.
    !!      - Note 1. and note 2. also apply in this case.
    !!  - if \c mode=2:
    !!      - The code performs as for mode 1, but the restriction \f$ z<20 \f$.
    !!      - For the evaluation of the continued fraction is not considered.
    !!
    !! Also, the parameter \c ipre can be used to select a different precision:
    !!  - For \c ipre=1, the precision is \f$10^{-12}\f$, taking
    !!    \f$\epsilon<10^{-12}\f$
    !!  - For \c ipre=2, the precision is \f$10^{-8}\f$, taking
    !!    \f$\epsilon<10^{-8}\f$
    !!
    !! where \f$\epsilon\f$ controls the accuracy.
    !!
    !! \warning Use \c mode 2 only if high \c m 's for \f$z>20\f$ are required.
    !! The evaluation of the cf may fail to converge for too high \c z 's.
    !!
    !! \note
```

```
    !!    -# For a precision of \f$10^{-12}\f$, if \f$z>5\f$ and \f$z/m>0.22\f$,
    !!    the code uses a series expansion for \c pl(0).\n
    !!    When \f$z<20\f$ and \f$z/m<0.22\f$ a continued fraction is applied.
    !!    -# For a precision of \f$10^{-8}\f$, if \f$z>5\f$ and \f$z/m>0.12\f$,
    !!    the code uses a series expansion for \c pl(0).\n
    !!    When \f$z<20\f$ and \f$z/m<0.12\f$ a continued fraction is applied.
    !!
    !! \return ierr
    integer function dtorh1(z,m,nmax,pl,ql,newn,mode,ipre) result(ierr)
        character(*), parameter :: rout_name = 'DTORH1'

        ! input / output
        real(dp), intent(in) :: z                                          !< (real) point at which toroidal harmonics are evaluated
        integer, intent(in) :: m                                           !< order of toroidal harmonics (\c m>0)
        integer, intent(in) :: nmax                                        !< maximum degree of toroidal harmonics (\c nmax>0)
        real(dp), intent(inout) :: pl(0:nmax)                              !< toroidal harmonics of first kind \f$P_{n-1/2}^m\left(z\right)\f$
        real(dp), intent(inout) :: ql(0:nmax)                              !< toroidal harmonics of second kind \f$Q_{n-1/2}^m\left(z\right)\f$
        integer, intent(inout) :: newn                                     !< maximum order of functions calculated when <tt> pl(nmax+1)>1/tiny </tt> with <tt>tiny=</tt>\f$10^{-290}\f$
        integer, intent(in), optional :: mode                              !< mode that controls output
        integer, intent(in), optional :: ipre                              !< precision

        ! local variables
        INTEGER nmaxp,mp,np,n,ical,i
        INTEGER :: mode_loc, ipre_loc
        REAL(dp) over,tinysq,qz,pisq,fl,cc,ar,gamma,fc,qdc1,qargu,&
            &ARGU1,DFACQS,FCP,DD,D1,QM0,DFAC3,DFAC4,PL0
        real(dp), allocatable :: pl_loc(:), ql_loc(:)                      ! local copy of pl and ql, possibly larger
        !REAL(dp) DPPI                                                     ! Is never used?
        !!!!
```

```
            ar=-1.
        END IF
        !!!!
```

```
            IF ((mp-1).NE.m) THEN
                ierr = 1
                err_msg = 'M IS TOO LARGE FOR MODE=1,2'
                chckerr(err_msg)
            END IF
        END IF
        !NMMAX=M                                                          ! Is never used?
        dfacqs=-(gamma/qlmm(m+1))*gamma/pi
        !!!!
```

```
        dn2=dn0+0.5_dp
        dn3=dn0-0.5_dp
        dn4=1._dp-m-m
        b=2._dp*dn1*z/dn2
        a=1._dp
        fc=tinysq
        c0=fc
        d0=0._dp
     81 d0=b+a*d0
        IF (dabs(d0).LT.tinysq) d0=tinysq
        c0=b+a/c0
        IF (dabs(c0).LT.tinysq) c0=tinysq
        d0=1._dp/d0
        delta=c0*d0
        fc=fc*delta
        mm=mm+1
        a=-(1.d0+dn4/(dn3+mm))
        b=dz2*(dn1+mm)/(dn2+mm)
        IF (mm.LT.1000000) THEN
            IF (dabs(delta-1.d0).GT.eps) GOTO 81
        END IF
        IF (mm.EQ.1000000) then
            ierr =1
            err_msg = 'CF CONVERGENCE FAILS FOR Z = '//trim(r2str(z))
            chckerr(err_msg)
        END IF
    end function frac

    integer function fracps(QZ,M,N,EPS,TINYSQ,FC) result(ierr)
        character(*), parameter :: rout_name = 'FRACPS'

        INTEGER m,n,mm
        REAL(dp) qz,eps,tinysq,fc,dn2,dz2,dm,b,a,c0,d0,delta

        ! initialize ierr
        ierr = 0

        mm=0
        dn2=n*n
        dz2=qz+qz
        dm=m-0.5_dp
        b=dz2*m
        a=dn2-dm*dm
        fc=tinysq
        c0=fc
        d0=0._dp
     82 d0=b+a*d0
        IF (dabs(d0).LT.tinysq) d0=tinysq
        c0=b+a/c0
        IF (dabs(c0).LT.tinysq) c0=tinysq
        d0=1._dp/d0
        delta=c0*d0
        fc=fc*delta
        mm=mm+1
        a=dn2-(mm+dm)*(mm+dm)
        b=dz2*(mm+m)
        IF (mm.LT.1000000) THEN
            IF (abs(delta-1._dp).GT.eps) GOTO 82
        END IF
        IF (mm.EQ.1000000) then
            ierr =1
            err_msg = 'CF CONVERGENCE FAILS FOR QZ = '//trim(r2str(qz))
            chckerr(err_msg)
        END IF
    end function fracps

    REAL(dp) function ellip2(ak)
        REAL(dp) ak
        REAL(dp) q

        q=(1._dp-ak)*(1._dp+ak)
        ellip2=drf(q)-(ak)**2*drd(q)/3.d0
    END FUNCTION ellip2

    REAL(dp) function ellip1(ak)
        REAL(dp) ak

        ellip1=drf((1._dp-ak)*(1._dp+ak))
    END FUNCTION ellip1


    !!!!
```

```
      REAL(dp) mu, s, xn, xndev
      REAL(dp) xnroot, y, yn, yndev, ynroot, zn, zndev, znroot
      LOGICAL first
      SAVE errtol,c1,c2,c3,first
      DATA first /.true./

      !***FIRST EXECUTABLE STATEMENT  DRF
      IF (first) THEN
          errtol = 1.e-8_dp
          c1 = 1.0_dp/24.0_dp
          c2 = 3.0_dp/44.0_dp
          c3 = 1.0_dp/14.0_dp
      ENDIF
      first = .false.
      drf = 0.0_dp
      xn = 0._dp
      yn = y
      zn = 1._dp

   30 mu = (xn+yn+zn)/3.0_dp
      xndev = 2.0_dp - (mu+xn)/mu
      yndev = 2.0_dp - (mu+yn)/mu
      zndev = 2.0_dp - (mu+zn)/mu
      epslon = max(abs(xndev),abs(yndev),abs(zndev))
      IF (epslon.LT.errtol) GO TO 40
      xnroot = sqrt(xn)
      ynroot = sqrt(yn)
      znroot = sqrt(zn)
      lamda = xnroot*(ynroot+znroot) + ynroot*znroot
      xn = (xn+lamda)*0.250_dp
      yn = (yn+lamda)*0.250_dp
      zn = (zn+lamda)*0.250_dp
      GO TO 30

   40 e2 = xndev*yndev - zndev*zndev
      e3 = xndev*yndev*zndev
      s  = 1.0_dp + (c1*e2-0.10_dp-c2*e3)*e2 + c3*e3
      drf = s/sqrt(mu)
  END FUNCTION drf


  REAL(dp) function drd (y)
      REAL(dp) epslon, errtol
      REAL(dp) c1, c2, c3, c4, ea, eb, ec, ed, ef, lamda
      REAL(dp) mu, power4, sigma, s1, s2, xn, xndev
      REAL(dp) xnroot, y, yn, yndev, ynroot,  zn, zndev, znroot
      LOGICAL first
      SAVE errtol, c1, c2, c3, c4, first
      DATA first /.true./

      !***FIRST EXECUTABLE STATEMENT  DRD
      IF (first) THEN
          errtol = 1.e-8_dp
          c1 = 3.0_dp/14.0_dp
          c2 = 1.0_dp/6.0_dp
          c3 = 9.0_dp/22.0_dp
          c4 = 3.0_dp/26.0_dp
      ENDIF
      first = .false.

      ! CALL ERROR HANDLER IF NECESSARY.

      drd = 0.0_dp

      xn = 0._dp
      yn = y
      zn = 1._dp
      sigma = 0.0_dp
      power4 = 1.0_dp

   30 mu = (xn+yn+3.0_dp*zn)*0.20_dp
      xndev = (mu-xn)/mu
      yndev = (mu-yn)/mu
      zndev = (mu-zn)/mu
      epslon = max(abs(xndev), abs(yndev), abs(zndev))
      IF (epslon.LT.errtol) GO TO 40
      xnroot = sqrt(xn)
      ynroot = sqrt(yn)
      znroot = sqrt(zn)
      lamda = xnroot*(ynroot+znroot) + ynroot*znroot
      sigma = sigma + power4/(znroot*(zn+lamda))
      power4 = power4*0.250_dp
      xn = (xn+lamda)*0.250_dp
      yn = (yn+lamda)*0.250_dp
      zn = (zn+lamda)*0.250_dp
      GO TO 30

```

```
 40 ea = xndev*yndev
    eb = zndev*zndev
    ec = ea - eb
    ed = ea - 6.0_dp*eb
    ef = ed + ec + ec
    s1 = ed*(-c1+0.250_dp*c3*ed-1.50_dp*c4*zndev*ef)
    s2 = zndev*(c2*ef+zndev*(-c3*ec+zndev*c4*ea))
    drd = 3.0_dp*sigma + power4*(1.0_dp+s1+s2)/(mu*sqrt(mu))
END FUNCTION drd

REAL(dp) function gammah(n,over)
    INTEGER n,i,j
    REAL(dp) over

    i=n
    j=2*i-1
    gammah=1._dp
 85 IF ((j.GE.1).AND.(gammah.LT.over)) THEN
        gammah=gammah*j/2._dp
        i=i-1
        j=2*i-1
        GOTO 85
    ENDIF
    IF (j.GT.1) THEN
        gammah=0._dp
    END IF
END FUNCTION gammah

REAL(dp) function factco(n,pl,m)
    INTEGER n,m,j
    REAL(dp) pl,x1,x2

    factco=1._dp/pl
    x1=m+0.5_dp
    x2=-m+0.5_dp
    j=n
86  IF (j.GE.0) THEN
        factco=factco*(j+x1)/(j+x2)
        j=j-1
        GOTO 86
    ENDIF
END FUNCTION factco

SUBROUTINE expan(Z,MODE,IPRE,OVER,QARGU,M,PL)
    INTEGER mode,ipre,m,k,i
    REAL(dp) z,over,qargu,pl,pisq,db,fl,cc,dz,gamma,ar,&
        &DFAC,DF1,A0,Z2I,DA1,DA2,DELTA,SUM,DCC,DCCP,DKK
    REAL(dp) preci(2)

    preci(1)=1.e-13_dp
    preci(2)=1.e-9_dp
    pisq=dsqrt(pi)
    db=2._dp*dlog(2._dp)
    fl=m/2.
    cc=abs(float(int(fl))-fl)
    IF (cc.LT.0.4_dp) THEN
        ar=1.
    ELSE
        ar=-1.
    END IF
    dz=1._dp
    DO 100 i=1,m
        dz=dz/qargu
100 CONTINUE
    IF (mode.EQ.0) THEN
        gamma=gammah(m,over)*ar*pisq
    ELSE
        gamma=ar
    END IF
    dfac=2._dp/pi*dz*gamma/pisq
    df1=dlog(2._dp*z)
    a0=1._dp/dsqrt(2._dp*z)
    z2i=1._dp/(z*z)
    delta=1._dp
    sum=0._dp
    k=0
    da2=factor(m,0)
    da1=db+psi(m,0)
 87 delta=(df1+da1)*da2*a0
    sum=sum+delta
    dcc=0.5_dp+m+2._dp*k
    dccp=dcc+1._dp
    dkk=k+1._dp
    da2=da2*dccp*dcc/(dkk*dkk)
    da1=da1+1._dp/dkk-1._dp/dccp-1._dp/dcc
    k=k+1
    a0=a0*0.25_dp*z2i
```

```
        IF (abs(delta/sum).GT.preci(ipre)) GOTO 87
        pl=sum*dfac
    END SUBROUTINE expan

    REAL(dp) function factor(m,k)
        INTEGER m,k,n,i,j
        REAL(dp) x1

        factor=1._dp
        IF (k.GE.1) THEN
        x1=m+0.5_dp
        n=2*k-1
        i=k
        j=n
     88 IF (j.GE.0) THEN
            factor=factor*(j+x1)/i/i
            j=j-1
            i=i-1
            IF (i.EQ.0) i=1
                GOTO 88
            ENDIF
        END IF
    END FUNCTION factor

    REAL(dp) function psi(m,k)
        INTEGER m,k,n,j,i
        REAL(dp) factr1,factr2
        psi=0._dp
        factr1=0._dp
        factr2=0._dp
        n=2*k+m
        j=1
        IF (k.GE.1) THEN
         89 IF (j.LE.k) THEN
                factr1=factr1+1._dp/j
                j=j+1
                GOTO 89
            ENDIF
        END IF
        i=1
     90 IF (i.LE.n) THEN
            factr2=factr2+1._dp/(2._dp*i-1._dp)
            i=i+1
            GOTO 90
        ENDIF
        IF (n.EQ.0) factr2=0._dp
        psi=factr1-2._dp*factr2
    END FUNCTION psi
end module
```

D.18 /Users/toonweyens/Code/PB3D/Modules/eq_ops.f90 File Reference

Interfaces and Types

- interface eq_ops::calc_eq

  Calculate the equilibrium quantities on a grid determined by straight field lines.
- interface eq_ops::print_output_eq

  Print equilibrium quantities to an output file:
- interface eq_ops::redistribute_output_eq

  Redistribute the equilibrium variables, but only the Flux variables are saved.
- interface eq_ops::calc_rzl

  Calculate R, Z & λ and derivatives in VMEC coordinates.
- interface eq_ops::calc_g_c

  Calculate the lower metric elements in the C(ylindrical) coordinate system.
- interface eq_ops::calc_g_v

  Calculate the lower metric coefficients in the equilibrium V(MEC) coordinate system.
- interface eq_ops::calc_g_h

Calculate the lower metric coefficients in the equilibrium H(ELENA) coordinate system.

- interface eq_ops::calc_g_f

  Calculate the metric coefficients in the F(lux) coordinate system.
- interface eq_ops::calc_jac_v

  Calculate $\mathcal{J}_V$, the jacobian in V(MEC) coordinates.
- interface eq_ops::calc_jac_h

  Calculate $\mathcal{J}_H$, the jacobian in HELENA coordinates.
- interface eq_ops::calc_jac_f

  Calculate $\mathcal{J}_F$, the jacobian in Flux coordinates.
- interface eq_ops::calc_t_vc

  Calculate $\overline{\overline{T}}_C^V$, the transformation matrix between C(ylindrical) and V(mec) coordinate system.
- interface eq_ops::calc_t_vf

  Calculate $\overline{\overline{T}}_V^F$, the transformation matrix between V(MEC) and F(lux) coordinate systems.
- interface eq_ops::calc_t_hf

  Calculate $\overline{\overline{T}}_H^F$, the transformation matrix between H(ELENA) and F(lux) coordinate systems.

Modules

- module eq_ops

  Operations on the equilibrium variables.

Functions/Subroutines

- integer function, public eq_ops::flux_q_plot (grid_eq, eq)

  Plots the flux quantities in the solution grid.
- integer function, public eq_ops::calc_derived_q (grid_eq, eq_1, eq_2)

  Calculates derived equilibrium quantities system.
- integer function, public eq_ops::calc_normalization_const ()

  Sets up normalization constants.
- subroutine, public eq_ops::normalize_input ()

  Normalize input quantities.
- integer function, public eq_ops::b_plot (grid_eq, eq_1, eq_2, rich_lvl, plot_fluxes, xyz)

  Plots the magnetic fields.
- integer function, public eq_ops::j_plot (grid_eq, eq_1, eq_2, rich_lvl, plot_fluxes, xyz)

  Plots the current.
- integer function, public eq_ops::kappa_plot (grid_eq, eq_1, eq_2, rich_lvl, xyz)

  Plots the curvature.
- integer function, public eq_ops::delta_r_plot (grid_eq, eq_1, eq_2, xyz, rich_lvl)

  *Plots **HALF** of the change in the position vectors for 2 different toroidal positions, which can correspond to a ripple.*
- integer function, public eq_ops::divide_eq_jobs (n_par_x, arr_size, n_div, n_div_max, n_par_x_base, range_name)

  Divides the equilibrium jobs.
- integer function, public eq_ops::calc_eq_jobs_lims (n_par_x, n_div)

  Calculate `eq_jobs_lims`.

Variables

- logical, public eq_ops::debug_calc_derived_q = .false.

  plot debug information for calc_derived_q()
- logical, public eq_ops::debug_j_plot = .false.

  plot debug information for j_plot()
- logical, public eq_ops::debug_create_vmec_input = .false.

  plot debug information for create_vmec_input()

D.19 eq_ops.f90

Go to the documentation of this file.

```
!-------------------------------------------------------------------------!
!> Operations on the equilibrium variables.
!-------------------------------------------------------------------------!
module eq_ops
#include <PB3D_macros.h>
    use str_utilities
    use output_ops
    use messages
    use num_vars, only: pi, dp, max_str_ln, max_deriv
    use grid_vars, only: grid_type
    use eq_vars, only: eq_1_type, eq_2_type
#if ldebug
    use num_utilities, only: check_deriv
#endif

    implicit none
    private
    public calc_eq, calc_derived_q, calc_normalization_const, normalize_input, &
        &print_output_eq, flux_q_plot, redistribute_output_eq, divide_eq_jobs, &
        &calc_eq_jobs_lims, calc_t_hf, b_plot, j_plot, kappa_plot, delta_r_plot
#if ldebug
    public debug_calc_derived_q, debug_create_vmec_input, debug_j_plot
#endif

    ! global variables
    integer :: fund_n_par                                                   !< fundamental interval width
    logical :: BR_normalization_provided(2)                                 ! used to export HELENA to VMEC
#if ldebug
    !> \ldebug
    logical :: debug_calc_derived_q = .false.                               !< plot debug information for calc_derived_q()
    !> \ldebug
    logical :: debug_j_plot = .false.                                       !< plot debug information for j_plot()
    !> \ldebug
    logical :: debug_create_vmec_input = .false.                            !< plot debug information for create_vmec_input()
#endif

    ! interfaces

    !> \public  Calculate the  equilibrium quantities  on a  grid determined  by
    !! straight field lines.
    !!
    !! This grid has the dimensions \c (n_par,loc_n_r).
    !!
    !! Optionally, for  \c eq_2, the  used variables  can be deallocated  on the
    !! fly, to limit memory usage.
    !!
    !! \return ierr
    interface calc_eq
        !> \public
        module procedure calc_eq_1
        !> \public
        module procedure calc_eq_2
    end interface

    !> \public Print equilibrium quantities to an output file:
    !!
    !!  - flux:
    !!      - \c pres_FD,
    !!      - \c q_saf_FD,
    !!      - \c rot_t_FD,
    !!      - \c flux_p_FD,
```

```
!!      - \c flux_t_FD,
!!      - \c rho,
!!      - \c S,
!!      - \c kappa_n,
!!      - \c kappa_g,
!!      - \c sigma
!!  - metric:
!!      - \c g_FD,
!!      - \c h_FD,
!!      - \c jac_FD
!!
!! If \c  rich_lvl is provided, \c  "_R_[rich_lvl]" is appended to  the data
!! name if it is \c >0 (only for \c eq_2).
!!
!! Optionally,  for \c  eq_2, it  can be  specified that  this is  a divided
!! parallel grid, corresponding  to the variable \c  eq_jobs_lims with index
!! \c eq_job_nr. In  this case, the total  grid size is adjusted  to the one
!! specified by \c eq_jobs_lims and the grid is written as a subset.
!! \note
!!  -# The equilibrium quantities are outputted in Flux coordinates.
!!  -#  The metric  equilibrium quantities  can be  deallocated on  the fly,
!!  which is useful  if this routine is followed by  a deallocation any way,
!!  so that memory usage does not almost double.
!!  -# \c print_output_eq_2  is only used by  HELENA now, as for  VMEC it is
!!  too slow since there are often multiple VMEC equilibrium jobs, while for
!!  HELENA this is explicitely forbidden.
!!
!! \return ierr
interface print_output_eq
    !> \public
    module procedure print_output_eq_1
    !> \public
    module procedure print_output_eq_2
end interface

!> \public  Redistribute  the  equilibrium  variables,  but  only  the  Flux
!! variables are saved.
!!
!! \see redistribute_output_grid()
!!
!! \return ierr
interface redistribute_output_eq
    !> \public
    module procedure redistribute_output_eq_1
    !> \public
    module procedure redistribute_output_eq_2
end interface

!> \public Calculate  \f$R\f$, \f$Z\f$  \& \f$\lambda\f$ and  derivatives in
!! VMEC coordinates.
!!
!! This is done at the grid points  given by the variables \c theta_E and \c
!! zeta_E (contained in \c trigon_factors) and at every normal point.
!!
!! The  derivatives are  indicated  by the  variable \c  deriv  which has  3
!! indices
!!
!! \note  There is  no  HELENA equivalent  because  for HELENA  simulations,
!! \f$R\f$  and \f$Z\f$  are not  necessary  for calculation  of the  metric
!! coefficients, and \f$\lambda\f$ does not exist.
!!
!! \see calc_trigon_factors()
!!
!! \return ierr
interface calc_rzl
    !> \public
    module procedure calc_rzl_ind
    !> \public
    module procedure calc_rzl_arr
end interface

!> \public  Calculate  the  lower  metric  elements  in  the  C(ylindrical)
!! coordinate system.
!!
!! This is done directly using the formula's in \cite Weyens3D
!!
!! \return ierr
interface calc_g_c
    !> \public
    module procedure calc_g_c_ind
    !> \public
    module procedure calc_g_c_arr
end interface

!> \public Calculate the lower metric coefficients in the equilibrium V(MEC)
!! coordinate system.
!!
```

```
    !! This is done using the metric coefficients in the C(ylindrical)
    !! coordinate system and the transformation matrices
    !!
    !! \note It is assumed that the lower order derivatives have been calculated
    !! already. If not, the results will be incorrect.
    !!
    !! \return ierr
    interface calc_g_v
        !> \public
        module procedure calc_g_v_ind
        !> \public
        module procedure calc_g_v_arr
    end interface

    !> \public Calculate the lower metric coefficients in the equilibrium
    !! H(ELENA) coordinate system.
    !!
    !! This is done using the HELENA output
    !!
    !! \return ierr
    interface calc_g_h
        !> \public
        module procedure calc_g_h_ind
        !> \public
        module procedure calc_g_h_arr
    end interface

    !> \public Calculate the metric coefficients in the F(lux) coordinate
    !! system.
    !!
    !! This is done using the metric coefficients in the equilibrium coordinate
    !! system and the transformation matrices.
    !!
    !! \return ierr
    interface calc_g_f
        !> \public
        module procedure calc_g_f_ind
        !> \public
        module procedure calc_g_f_arr
    end interface

    !> \public Calculate \f$\mathcal{J}_\text{V}\f$, the jacobian in V(MEC)
    !! coordinates.
    !!
    !! This is done using VMEC output.
    !!
    !! \return ierr
    interface calc_jac_v
        !> \public
        module procedure calc_jac_v_ind
        !> \public
        module procedure calc_jac_v_arr
    end interface

    !> \public Calculate \f$\mathcal{J}_\text{H}\f$, the jacobian in HELENA
    !! coordinates.
    !!
    !! This is done directly from
    !!  \f[\mathcal{J}_\text{H} = q \frac{R^2}{F} \f]
    !!
    !! \note It is assumed that the lower order derivatives have been calculated
    !! already. If not, the results will be incorrect.
    !!
    !! \return ierr
    interface calc_jac_h
        !> \public
        module procedure calc_jac_h_ind
        !> \public
        module procedure calc_jac_h_arr
    end interface

    !> \public Calculate \f$\mathcal{J}_\text{F}\f$, the jacobian in Flux
    !! coordinates.
    !!
    !! This is done directly from
    !!  \f[ \mathcal{J}_\text{F} =
    !!      \text{det}\left(\overline{\text{T}}_\text{F}^\text{E}\right)
    !!      \mathcal{J}_\text{E} \f]
    !!
    !! \note It is assumed that the lower order derivatives have been calculated
    !! already. If not, the results will be incorrect.
    !!
    !! \return ierr
    interface calc_jac_f
        !> \public
        module procedure calc_jac_f_ind
        !> \public
```

```
        module procedure calc_jac_f_arr
    end interface

    !> \public   Calculate    \f$\overline{\text{T}}_\text{C}^\text{V}\f$,   the
    !! transformation matrix between C(ylindrical) and V(mec) coordinate system.
    !!
    !! This is done directly using the formula's in \cite Weyens3D
    !!
    !! \return ierr
    interface calc_t_vc
        !> \public
        module procedure calc_t_vc_ind
        !> \public
        module procedure calc_t_vc_arr
    end interface

    !> \public   Calculate    \f$\overline{\text{T}}_\text{V}^\text{F}\f$,   the
    !! transformation matrix between V(MEC) and F(lux) coordinate systems.
    !!
    !! This is done directly using the formula's in \cite Weyens3D
    !!
    !! \return ierr
    interface calc_t_vf
        !> \public
        module procedure calc_t_vf_ind
        !> \public
        module procedure calc_t_vf_arr
    end interface

    !> \public   Calculate    \f$\overline{\text{T}}_\text{H}^\text{F}\f$,   the
    !! transformation matrix between H(ELENA) and F(lux) coordinate systems.
    !!
    !! This is done directly using the formula's in \cite Weyens3D
    !!
    !! \return ierr
    interface calc_t_hf
        !> \public
        module procedure calc_t_hf_ind
        !> \public
        module procedure calc_t_hf_arr
    end interface

contains
    !> \private flux version
    integer function calc_eq_1(grid_eq,eq) result(ierr)
        use num_vars, only: eq_style, rho_style, use_normalization, &
            &use_pol_flux_e, use_pol_flux_f
        use eq_vars, only: rho_0
        use num_utilities, only: derivs
        use eq_utilities, only: calc_f_derivs

        character(*), parameter :: rout_name = 'calc_eq_1'

        ! input / output
        type(grid_type), intent(inout) :: grid_eq                               !< equilibrium grid
        type(eq_1_type), intent(inout) :: eq                                    !< flux equilibrium variables

        ! local variables
        character(len=max_str_ln) :: err_msg                                    ! error message
        character(len=max_str_ln) :: file_name                                  ! file name
        character(len=max_str_ln) :: plot_title                                 ! plot title

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Start setting up flux equilibrium quantities')

        call lvl_ud(1)

        ! create equilibrium
        call eq%init(grid_eq)

        ! choose which equilibrium style is being used:
        !   1:  VMEC
        !   2:  HELENA
        select case (eq_style)
            case (1)                                                            ! VMEC
                call calc_flux_q_vmec()
            case (2)                                                            ! HELENA
                call calc_flux_q_hel()
        end select

        ! tests
        err_msg = 'Check out the FAQ at &
            &https://pb3d.github.io/Doxygen/html/page_faq.html'

```

```
        ! check whether the coordinate is increasing
        if (grid_eq%r_F(grid_eq%n(3)) .le. grid_eq%r_F(1)) then
            file_name = 'r_F'
            plot_title = 'Flux normal coordinate'
            call print_ex_2d(plot_title,file_name,grid_eq%r_F,draw=.false.)
            call draw_ex([plot_title],file_name,1,1,.false.)
            ierr = 3
            call writo('The code has not been tested satisfactorily for &
                &decreasing normal coordinate. See plot '// trim(file_name),&
                &alert=.true.)
            chckerr(err_msg)
        end if

        ! check whether there are reversed shear regions
        if (maxval(eq%q_saf_E(:,1))*minval(eq%q_saf_E(:,1)) .lt. 0._dp) then
            file_name = 'Dq_saf'
            plot_title = 'derivative of safety factor'
            call print_ex_2d(plot_title,file_name,eq%q_saf_E(:,1),draw=.false.)
            call draw_ex([plot_title],file_name,1,1,.false.)
            ierr = 3
            call writo('The code has to be adapted for reversed-shear regions &
                &still. See plot ' // trim(file_name),alert=.true.)
            chckerr(err_msg)
        end if

        ! take local variables
        grid_eq%loc_r_E = grid_eq%r_E(grid_eq%i_min:grid_eq%i_max)
        grid_eq%loc_r_F = grid_eq%r_F(grid_eq%i_min:grid_eq%i_max)

        ! Transform flux equilibrium E into F derivatives
        ierr = calc_f_derivs(grid_eq,eq)
        chckerr('')

        ! Calculate particle density rho
        ! choose which density style is being used:
        !   1:  constant, equal to rho_0
        select case (rho_style)
            case (1)                                                    ! arbitrarily constant (normalized value)
                eq%rho = rho_0
            case default
                err_msg = 'No density style associated with '//&
                    &trim(i2str(rho_style))
                ierr = 1
                chckerr(err_msg)
        end select
        ! normalize rho if necessary
        if (use_normalization) eq%rho = eq%rho/rho_0

        call lvl_ud(-1)

        call writo('Done setting up flux equilibrium quantities')
    contains
        ! VMEC version
        ! The VMEC normal coord. is  the toroidal (or poloidal) flux, normalized
        ! wrt.  to  the  maximum  flux,  equidistantly,  so  the  step  size  is
        ! 1/(n(3)-1)
        !> \private
        subroutine calc_flux_q_vmec()
            use vmec_vars, only: rot_t_v, q_saf_v, flux_t_v, flux_p_v, pres_v
            use eq_vars, only: max_flux_e

            ! copy flux variables
            eq%flux_p_E = flux_p_v(grid_eq%i_min:grid_eq%i_max,:)
            eq%flux_t_E = flux_t_v(grid_eq%i_min:grid_eq%i_max,:)
            eq%pres_E = pres_v(grid_eq%i_min:grid_eq%i_max,:)
            eq%q_saf_E = q_saf_v(grid_eq%i_min:grid_eq%i_max,:)
            eq%rot_t_E = rot_t_v(grid_eq%i_min:grid_eq%i_max,:)

            ! max flux and  normal coord. of eq grid  in Equilibrium coordinates
            ! (uses poloidal flux by default)
            if (use_pol_flux_e) then
                grid_eq%r_E = flux_p_v(:,0)/max_flux_e
            else
                grid_eq%r_E = flux_t_v(:,0)/max_flux_e
            end if

            ! max flux and normal coord. of eq grid in Flux coordinates
            if (use_pol_flux_f) then
                grid_eq%r_F = flux_p_v(:,0)/(2*pi)                      ! psi_F = flux_p/2pi
            else
                grid_eq%r_F = - flux_t_v(:,0)/(2*pi)                    ! psi_F = flux_t/2pi, conversion VMEC LH ->
    PB3D RH
            end if
        end subroutine calc_flux_q_vmec

        ! HELENA version
        ! The HELENA normal coord. is the poloidal flux divided by 2pi
```

```
        !> \private
        subroutine calc_flux_q_hel()
            use helena_vars, only: q_saf_h, rot_t_h, flux_p_h, flux_t_h, pres_h
            use num_utilities, only: calc_int

            ! copy flux variables
            eq%flux_p_E = flux_p_h(grid_eq%i_min:grid_eq%i_max,:)
            eq%flux_t_E = flux_t_h(grid_eq%i_min:grid_eq%i_max,:)
            eq%pres_E = pres_h(grid_eq%i_min:grid_eq%i_max,:)
            eq%q_saf_E = q_saf_h(grid_eq%i_min:grid_eq%i_max,:)
            eq%rot_t_E = rot_t_h(grid_eq%i_min:grid_eq%i_max,:)

            ! max flux and  normal coord. of eq grid  in Equilibrium coordinates
            ! (uses poloidal flux by default)
            grid_eq%r_E = flux_p_h(:,0)/(2*pi)

            ! max flux and normal coord. of eq grid in Flux coordinates
            if (use_pol_flux_f) then
                grid_eq%r_F = flux_p_h(:,0)/(2*pi)                              ! psi_F = flux_p/2pi
            else
                grid_eq%r_F = flux_t_h(:,0)/(2*pi)                              ! psi_F = flux_t/2pi
            end if
        end subroutine calc_flux_q_hel
    end function calc_eq_1
    !> \private metric version
    integer function calc_eq_2(grid_eq,eq_1,eq_2,dealloc_vars) result(ierr)
        use num_vars, only: eq_style, export_hel, tol_zero
        use num_utilities, only: derivs, c
        use eq_utilities, only: calc_inv_met, calc_f_derivs
        use vmec_utilities, only: calc_trigon_factors
#if ldebug
        use num_vars, only: ltest
        use input_utilities, only: get_log, pause_prog
        use helena_ops, only: test_metrics_h, test_harm_cont_h
        use helena_vars, only: chi_h
#endif

        character(*), parameter :: rout_name = 'calc_eq_2'

        ! input / output
        type(grid_type), intent(inout) :: grid_eq                               !< equilibrium grid
        type(eq_1_type), intent(in) :: eq_1                                     !< metric equilibrium variables
        type(eq_2_type), intent(inout) :: eq_2                                  !< metric equilibrium variables
        logical, intent(in), optional :: dealloc_vars                           !< deallocate variables on the fly after writing

        ! local variables
        integer :: id, jd, kd                                                   ! counters
        integer :: pmone                                                        ! plus or minus one
        logical :: dealloc_vars_loc                                             ! local dealloc_vars
        character(len=max_str_ln) :: err_msg                                    ! error message

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Start setting up metric equilibrium quantities')

        call lvl_ud(1)

        ! set up local dealloc_vars
        dealloc_vars_loc = .false.
        if (present(dealloc_vars)) dealloc_vars_loc = dealloc_vars

        ! create metric equilibrium variables
        call eq_2%init(grid_eq)

        ! do some preparations depending on equilibrium style used
        !   1:  VMEC
        !   2:  HELENA
        select case (eq_style)
            case (1)                                                            ! VMEC
                ! calculate  the   cylindrical  variables   R,  Z  and   λ  and
                ! derivatives if not yet done
                call writo('Calculate R, Z, λ, ...')
                if (.not.allocated(grid_eq%trigon_factors)) then
                    ierr = calc_trigon_factors(grid_eq%theta_E,grid_eq%zeta_E,&
                        &grid_eq%trigon_factors)
                    chckerr('')
                end if
                do id = 0,max_deriv+1
                    ierr = calc_rzl(grid_eq,eq_2,derivs(id))
                    chckerr('')
                end do
            case (2)                                                            ! HELENA
                ! do nothing
        end select
```

```

        ! Calcalations depending on equilibrium style being used:
        !   1:  VMEC
        !   2:  HELENA
        select case (eq_style)
            case (1)                                                        ! VMEC
                ! calculate the metrics in the cylindrical coordinate system
                call writo('Calculate g_C')                                 ! h_C is not necessary
                do id = 0,max_deriv
                    ierr = calc_g_c(eq_2,derivs(id))
                    chckerr('')
                end do

                ! calculate the transformation matrix C(ylindrical) -> V(MEC)
                call writo('Calculate T_VC')
                do id = 0,max_deriv
                    ierr = calc_t_vc(eq_2,derivs(id))
                    chckerr('')
                end do

                ! calculate the metric factors in the VMEC coordinate system
                call writo('Calculate g_V')
                do id = 0,max_deriv
                    ierr = calc_g_v(eq_2,derivs(id))
                    chckerr('')
                end do

                ! calculate the jacobian in the VMEC coordinate system
                call writo('Calculate jac_V')
                do id = 0,max_deriv
                    ierr = calc_jac_v(grid_eq,eq_2,derivs(id))
                    chckerr('')
                end do

#if ldebug
                if (ltest) then
                    call writo('Test calculation of g_V?')
                    if(get_log(.false.)) then
                        ierr = test_g_v(grid_eq,eq_2)
                        chckerr('')
                        call pause_prog
                    end if
                    call writo('Test calculation of jac_V?')
                    if(get_log(.false.)) then
                        ierr = test_jac_v(grid_eq,eq_2)
                        chckerr('')
                        call pause_prog
                    end if
                end if
#endif

                ! calculate the transformation matrix V(MEC) -> F(lux)
                call writo('Calculate T_VF')
                do id = 0,max_deriv
                    ierr = calc_t_vf(grid_eq,eq_1,eq_2,derivs(id))
                    chckerr('')
                end do

                ! set up plus minus one
                pmone = -1                                                  ! conversion VMEC LH -> RH coord. system

                ! possibly deallocate
                if (dealloc_vars_loc) then
                    deallocate(eq_2%g_C)
                end if
            case (2)                                                        ! HELENA
#if ldebug
                ! Check whether poloidal grid is indeed chi_H
                ! If not,  the boundary  conditions used in  the splines  of the
                ! following routines are not correct.
                ! A  solution to  this problem  would be  to set  up the  HELENA
                ! variables  in  a  full  periodic  grid,  even  when  they  are
                ! top-bottom symmetric.
                do kd = 1,grid_eq%loc_n_r
                    do jd = 1,grid_eq%n(2)
                        do id = 1,grid_eq%n(1)
                            if (abs(grid_eq%theta_E(id,jd,kd)-chi_h(id)).gt.&
                                &tol_zero) then
                                ierr = 1
                                err_msg = 'theta_E is not identical to chi_H'
                                chckerr(err_msg)
                            end if
                            if (abs(grid_eq%zeta_E(id,jd,kd)-0._dp).gt.&
                                &tol_zero) then
                                ierr = 1
                                err_msg = 'zeta_E is not identical to 0'
                                chckerr(err_msg)
```

```
                            end if
                        end do
                    end do
                end do

                if (ltest) then
                    call writo('Test consistency of metric factors?')
                    if(get_log(.false.,ind=.true.)) then
                        ierr = test_metrics_h()
                        chckerr('')
                        call pause_prog(ind=.true.)
                    end if

                    call writo('Test harmonic content of HELENA output?')
                    if(get_log(.false.,ind=.true.)) then
                        ierr = test_harm_cont_h()
                        chckerr('')
                        call pause_prog(ind=.true.)
                    end if
                end if
#endif

                ! calculate the jacobian in the HELENA coordinate system
                call writo('Calculate jac_H')
                do id = 0,max_deriv
                    ierr = calc_jac_h(grid_eq,eq_1,eq_2,derivs(id))
                    chckerr('')
                end do

                ! calculate the metric factors in the HELENA coordinate system
                call writo('Calculate g_H')
                do id = 0,max_deriv
                    ierr = calc_g_h(grid_eq,eq_2,derivs(id))
                    chckerr('')
                end do

                ! calculate the inverse h_H of the metric factors g_H
                call writo('Calculate h_H')
                do id = 0,max_deriv
                    ierr = calc_inv_met(eq_2%h_E,eq_2%g_E,derivs(id))
                    chckerr('')
                end do

#if ldebug
                if (ltest) then
                    call writo('Test calculation of D1 D2 h_H?')
                    if(get_log(.false.)) then
                        ierr = test_d12h_h(grid_eq,eq_2)
                        chckerr('')
                        call pause_prog
                    end if
                end if
#endif

                ! export for VMEC port
                if (export_hel) then
                    call writo('Exporting HELENA equilibrium for VMEC porting')
                    call lvl_ud(1)
                    ierr = create_vmec_input(grid_eq,eq_1)
                    chckerr('')
                    call lvl_ud(-1)
                    call writo('Done exporting')
                end if

                ! calculate the transformation matrix H(ELENA) -> F(lux)
                call writo('Calculate T_HF')
                do id = 0,max_deriv
                    ierr = calc_t_hf(grid_eq,eq_1,eq_2,derivs(id))
                    chckerr('')
                end do

                ! set up plus minus one
                pmone = 1

                ! possibly deallocate
                if (dealloc_vars_loc) then
                    deallocate(eq_2%h_E)
                end if
        end select

#if ldebug
        if (ltest) then
            call writo('Test calculation of T_EF?')
            if(get_log(.false.)) then
                ierr = test_t_ef(grid_eq,eq_1,eq_2)
                chckerr('')
                call pause_prog
```

```
            end if
        end if
#endif

        ! calculate  the  inverse of  the transformation  matrix T_EF
        call writo('Calculate T_FE')
        do id = 0,max_deriv
            ierr = calc_inv_met(eq_2%T_FE,eq_2%T_EF,derivs(id))
            chckerr('')
            ierr = calc_inv_met(eq_2%det_T_FE,eq_2%det_T_EF,derivs(id))
            chckerr('')
        end do

        ! calculate the metric factors in the Flux coordinate system
        call writo('Calculate g_F')
        do id = 0,max_deriv
            ierr = calc_g_f(eq_2,derivs(id))
            chckerr('')
        end do

        ! calculate the inverse h_F of the metric factors g_F
        call writo('Calculate h_F')
        do id = 0,max_deriv
            ierr = calc_inv_met(eq_2%h_F,eq_2%g_F,derivs(id))
            chckerr('')
        end do

        ! calculate the jacobian in the Flux coordinate system
        call writo('Calculate jac_F')
        do id = 0,max_deriv
            ierr = calc_jac_f(eq_2,derivs(id))
            chckerr('')
        end do

        ! limit Jacobians to small value to avoid infinities
        eq_2%jac_F(:,:,:,0,0,0) = sign(&
            &max(tol_zero,abs(eq_2%jac_F(:,:,:,0,0,0))),eq_2%jac_F(:,:,:,0,0,0))
        eq_2%jac_E(:,:,:,0,0,0) = sign(&
            &max(tol_zero,abs(eq_2%jac_E(:,:,:,0,0,0))),eq_2%jac_E(:,:,:,0,0,0))

        ! possibly deallocate
        if (dealloc_vars_loc) then
            deallocate(eq_2%g_E)
        end if

        ! Transform metric equilibrium E into F derivatives
        ierr = calc_f_derivs(eq_2)
        chckerr('')

        ! possibly deallocate
        if (dealloc_vars_loc) then
            deallocate(eq_2%g_F,eq_2%h_F,eq_2%jac_F)
            deallocate(eq_2%det_T_EF,eq_2%det_T_FE)
        end if

        ! Calculate derived metric quantities
        ierr = calc_derived_q(grid_eq,eq_1,eq_2)
        chckerr('')

#if ldebug
        if (ltest) then
            call writo('Test Jacobian in Flux coordinates?')
            if(get_log(.false.)) then
                ierr = test_jac_f(grid_eq,eq_1,eq_2)
                chckerr('')
                call pause_prog
            end if
            call writo('Test calculation of B_F?')
            if(get_log(.false.)) then
                ierr = test_b_f(grid_eq,eq_1,eq_2)
                chckerr('')
                call pause_prog
            end if
            call writo('Test consistency with given pressure?')
            if(get_log(.false.)) then
                ierr = test_p(grid_eq,eq_1,eq_2)
                chckerr('')
                call pause_prog
            end if
        end if
#endif

        call lvl_ud(-1)

        call writo('Done setting up metric equilibrium quantities')
    end function calc_eq_2

```

```
    !> Creates a VMEC input file.
    !!
    !! Optionally, a perturbation  can be added: Either the  displacement of the
    !! plasma position  can be  described (\c  pert_style 1),  or ripple  in the
    !! toroidal magnetic  field (\c  pert_style 2), with  a fixed  toroidal mode
    !! number.
    !!
    !! Both perturbation styles can have various prescription types:
    !!  -# file with Fourier modes in the geometrical angular coordinate
    !!  -# same but manually
    !!  -#  file with  perturbation  from a  2-D map  in  the geometric  angular
    !!  coordinate.
    !!
    !! For  \c pert_style  2,  a file  has  to be  provided  that describes  the
    !! translation between  position perturbation and magnetic  perturbation for
    !! curves of constant  geometrical angle. This file can be  generated for an
    !! already existing  ripple case using POST  with <tt>--compare_tor_pos</tt>
    !! with <tt>n_zeta_plot = 3</tt> and \c min_theta_plot and \c max_theta_plot
    !! indicating half a ripple period.
    !!
    !! The output from  this VMEC run can  then be used to  iteratively create a
    !! new  file  to  translate  toroidal  magnetic  field  ripple  to  position
    !! perturbation.
    !!
    !! \note Meaning of the indices of \c B_F, \c B_F_dum:
    !!  - <tt>(pol modes, cos/sin)</tt> for \c B_F_dum
    !!  - <tt>(tor_modes, pol modes, cos/sin (m theta), R/Z)</tt> for \c B_F
    integer function create_vmec_input(grid_eq,eq_1) result(ierr)
        use eq_vars, only: pres_0, r_0, psi_0, b_0, vac_perm
        !use eq_vars, only: max_flux_E
        use grid_vars, only: n_r_eq
        use grid_utilities, only: nufft, calc_xyz_grid
        use helena_vars, only: nchi, r_h, z_h, ias, rbphi_h, flux_p_h, &
            &flux_t_h, chi_h, rmtog_h, bmtog_h
        use x_vars, only: min_r_sol, max_r_sol
        use input_utilities, only: pause_prog, get_log, get_int, get_real
        use num_utilities, only: gcd, bubble_sort, order_per_fun, &
            &shift_f, calc_int, spline
        use num_vars, only: eq_name, hel_pert_i, hel_export_i, &
            &norm_disc_prec_eq, prop_b_tor_i, tol_zero, mu_0_original, &
            &use_normalization
        use files_utilities, only: skip_comment, count_lines
        use ezspline_obj
        use ezspline
#if ldebug
        use num_vars, only: ltest
#endif

        character(*), parameter :: rout_name = 'create_VMEC_input'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                          !< equilibrium grid varibles
        type(eq_1_type), intent(in) :: eq_1                             !< flux equilibrium quantities

        ! local variables
        type(ezspline2_r8) :: f_spl                                     ! spline object for interpolation
        integer :: id, jd, kd, ld                                       ! counters
        integer :: n_b                                                  ! nr. of points in Fourier series
        integer :: nfp                                                  ! scale factor for toroidal mode numbers
        integer :: tot_nr_pert                                          ! total number of perturbations combinations
    (N,M)
        integer :: nr_n                                                 ! number of different N in perturbation
        integer :: id_n_0                                               ! index where zero N is situated in B_F
        integer :: n_loc, m_loc                                         ! local n and m of perturbation
        integer :: pert_map_n_loc                                       ! n_loc for perturbation map
        integer :: n_id                                                 ! index in bundled n_pert
        integer :: n_pert_map(2)                                        ! number of points in perturbation map
        integer :: plot_dim(2)                                          ! plot dimensions
        integer :: rec_min_m                                            ! recommended minimum of poloidal modes
        integer :: pert_style                                           ! style of perturbation (1: r, 2: B_tor)
        integer :: pert_type                                            ! type of perturbation prescription
        integer :: max_n_b_output                                       ! max. nr. of modes written in output file
    (constant in VMEC)
        integer :: n_prop_b_tor                                         ! n of angular poins in prop_B_tor
        integer :: nr_m_max                                             ! maximum nr. of poloidal mode numbers
        integer :: ncurr                                                ! ncurr VMEC flag (0: prescribe iota, 1:
    prescribe tor. current)
        integer :: bcs(2,2)                                             ! boundary conditions
        integer :: m_range(2)                                           ! range in poloidal modes
        integer, allocatable :: n_pert(:)                               ! tor. mode numbers and pol. mode numbers
    for each of them
        integer, allocatable :: n_pert_copy(:)                          ! copy of n_pert, for sorting
        integer, allocatable :: piv(:)                                  ! pivots for sorting
        character(len=1) :: pm(3)                                       ! "+ " or "-"
        character(len=8) :: flux_name(2)                                ! "poloidal" or "toroidal"
        character(len=6) :: path_prefix = '../../'                      ! prefix of path
        character(len=max_str_ln) :: hel_pert_file_name                 ! name of perturbation file
```

```
        character(len=max_str_ln) :: err_msg                                    ! error message
        character(len=max_str_ln) :: file_name                                  ! name of file
        character(len=max_str_ln) :: plot_name(2)                               ! name of plot file
        character(len=max_str_ln) :: plot_title(2)                              ! name of plot
        character(len=max_str_ln) :: prop_b_tor_file_name                       ! name of B_tor proportionality file
        real(dp) :: eq_vert_shift                                               ! vertical shift in equilibrium
        real(dp) :: pert_map_vert_shift                                         ! vertical shift in perturbation map
        real(dp) :: max_pert_on_axis                                            ! maximum perturbation on axis (theta = 0)
        real(dp) :: m_tol = 1.e-7_dp                                            ! tolerance for Fourier mode strength
        real(dp) :: delta_loc(2)                                                ! local delta
        real(dp) :: plot_lims(2,2)                                              ! limits of plot dims [pi]
        real(dp) :: norm_b_h                                                    ! normalization for R and Z Fourier modes
        real(dp) :: mult_fac                                                    ! global multiplication factor
        real(dp) :: prop_b_tor_smooth                                           ! smoothing of prop_B_tor_interp via Fourier transform
        real(dp) :: rz_b_0(2)                                                   ! origin of R and Z of boundary
        real(dp) :: s_vj(99)                                                    ! normal coordinate s for writing
        real(dp) :: pres_vj(99,0:1)                                             ! pressure for writing
        real(dp) :: rot_t_v(99)                                                 ! rotational transform for writing
        real(dp) :: f_vj(99)                                                    ! F for writing
        real(dp) :: ffp_vj(99)                                                  ! FF' for writing
        real(dp) :: flux_j(99)                                                  ! normalized flux for writing
        real(dp) :: q_saf_vj(99)                                                ! q_saf for writing
        real(dp) :: i_tor_v(99)                                                 ! enclosed toroidal current for VMEC
        real(dp) :: i_tor_j(99)                                                 ! enclosed toroidal current for JOREK
        real(dp) :: norm_j(3)                                                   ! normalization factors for Jorek (R_geo, Z_geo, F0)
        real(dp), allocatable :: psi_t(:)                                       ! normalized toroidal flux
        real(dp), allocatable :: ffp(:)                                         ! FF' in total grid
        real(dp), allocatable :: i_tor(:)                                       ! toroidal current within flux surfaces
        real(dp), allocatable :: norm_transf(:,:)                               ! transformation of normal coordinates between MISHKA and VMEC or JOREK
        real(dp), allocatable :: i_tor_int(:)                                   ! integrated I_tor
        real(dp), allocatable :: rrint(:)                                       ! R^2 integrated poloidally
        real(dp), allocatable :: rrint_loc(:)                                   ! local RRint
        real(dp), allocatable :: r_plot(:,:,:)                                  ! R for plotting of ripple map
        real(dp), allocatable :: z_plot(:,:,:)                                  ! Z for plotting of ripple map
        real(dp), allocatable :: pert_map_r(:)                                  ! R of perturbation map
        real(dp), allocatable :: pert_map_z(:)                                  ! Z of perturbation map
        real(dp), allocatable :: pert_map(:,:)                                  ! perturbation map
        real(dp), allocatable :: pert_map_interp(:)                             ! interpolated perturbation from map
        real(dp), allocatable :: pert_map_interp_f(:,:)                         ! Fourier coefficients of pert_map_interp
        real(dp), allocatable :: r_h_loc(:,:)                                   ! local R_H
        real(dp), allocatable :: z_h_loc(:,:)                                   ! local Z_H
        real(dp), allocatable :: delta(:,:,:)                                   ! amplitudes of perturbations (N,M,c/s)
        real(dp), allocatable :: delta_copy(:,:,:)                              ! copy of delta, for sorting
        real(dp), allocatable :: bh_0(:,:)                                      ! R and Z at unperturbed bounday
        real(dp), allocatable :: b_f(:,:,:,:)                                   ! cos and sin Fourier components
        real(dp), allocatable :: b_f_copy(:,:,:,:)                              ! copy of B_F
        real(dp), allocatable :: b_f_dum(:,:)                                   ! dummy B_F
        real(dp), allocatable :: b_f_dum2(:,:)                                  ! other dummy B_F
        real(dp), allocatable :: b_f_dum3(:,:)                                  ! other dummy B_F
        real(dp), allocatable :: theta_b(:)                                     ! geometric pol. angle
        real(dp), allocatable :: theta_geo(:,:,:)                               ! geometric pol. angle, for plotting
        real(dp), allocatable :: prop_b_tor(:,:)                                ! proportionality between delta B_tor and delta_norm
        real(dp), allocatable :: prop_b_tor_ord(:,:)                            ! ordered prop_B_tor
        real(dp), allocatable :: prop_b_tor_interp(:)                           ! proportionality between delta B_tor and delta_norm
        real(dp), allocatable :: prop_b_tor_interp_f(:,:)                       ! Fourier modes of prop_B_tor_interp
        real(dp), allocatable :: xyz_plot(:,:,:,:)                              ! plotting X, Y and Z
        logical :: zero_n_pert                                                  ! there is a perturbation with N = 0
        logical :: pert_eq                                                      ! whether equilibrium is perturbed
        logical :: stel_sym                                                     ! whether there is stellarator symmetry
        logical :: change_max_n_b_output                                        ! whether to change max_n_B_output
        logical :: found                                                        ! whether something was found
#if ldebug
        real(dp), allocatable :: bh_0_alt(:,:)                                  ! reconstructed R and Z
#endif

        ! initialize ierr
        ierr = 0

        ! test if full range
        if (min_r_sol.gt.0 .or. max_r_sol.lt.1) then
            ierr = 1
            err_msg = 'This routine should be run with the full normal &
                &range!'
            chckerr(err_msg)
        end if

        ! test if normalization constants chosen
        if (.not.all(br_normalization_provided,1)) &
            &call writo('No normalization factors were provided. &
            &Are you sure this is correct?',warning=.true.)

        ! set up local R_H and Z_H
```

```
        allocate(r_h_loc(nchi,n_r_eq))
        allocate(z_h_loc(nchi,n_r_eq))
        r_h_loc = r_h
        z_h_loc = z_h
        if (use_normalization) then
            r_h_loc = r_h*r_0
            z_h_loc = z_h*r_0
        end if

        ! ask for vertical shift
        ! Note: HELENA automatically shifts it zo Zvac = 0
        call writo('Was the equilibrium shifted vertically?')
        eq_vert_shift = 0._dp
        if (get_log(.false.)) then
            call writo('How much higher [m] should the &
                &equilibrium be in the real world?')
            eq_vert_shift = get_real()
            call writo('The equilibrium will be shifted by '//&
                &trim(r2strt(eq_vert_shift))//'m to compensate for this')
            z_h_loc = z_h_loc + eq_vert_shift
        end if

        ! set up F, FF' and <R^2>
        allocate(ffp(n_r_eq))
        allocate(rrint(n_r_eq))
        allocate(rrint_loc(nchi))                                   ! includes overlap, as necessary for integration
        ierr = spline(flux_p_h(:,0)/(2*pi),rbphi_h(:,0)**2*0.5_dp,&
            &flux_p_h(:,0)/(2*pi),ffp,ord=norm_disc_prec_eq,deriv=1)
        chckerr('')
        do kd = 1,n_r_eq
            ierr = calc_int(r_h(:,kd)**2,chi_h,rrint_loc)
            chckerr('')
            rrint(kd) = rrint_loc(nchi)
        end do
        if (ias.eq.0) rrint = 2*rrint

        ! set up toroidal current:
        ! This is the  toroidal current between neighbouring  flux surfaces (see
        ! first commentary in VMEC2013/Sources/General/add_fluxes.f90:
        !   <J^zeta J>  = - [2pi FF'/mu_0 <J/R^2> + p'<J>]
        !               = - [2pi F'q/mu_0 + p' <R^2>q/F] ,
        ! which all  have units  1/m. Here, <.>  means integration  in a
        ! full poloidal period.
        ! As VMEC  wants the toroidal current  within two infintesimally
        ! separated  flux  surface as  a  function  of the  VMEC  normal
        ! coordinate, above has to be multiplied by
        !   dpsi_PB3D/dpsi_V = psi_tor(edge)/(2pi q)
        ! Finally, there is  an extra factor -1 because  the toroidal coordinate
        ! in VMEC is oposite.
        ! For JOREK,  the same  is done  with the poloidal  flux instead  of the
        ! toroidal and without the change of sign.
        allocate(i_tor(n_r_eq))
        allocate(norm_transf(n_r_eq,2))
        norm_transf(:,1) = -flux_t_h(n_r_eq,0)/(2*pi*eq_1%q_saf_E(:,0))
        norm_transf(:,2) = flux_p_h(n_r_eq,0)/(2*pi)
        i_tor = - (eq_1%pres_E(:,1)*rrint*eq_1%q_saf_E(:,0)/rbphi_h(:,0) + &
            &2*pi*ffp*eq_1%q_saf_E(:,0)/(rbphi_h(:,0)*vac_perm))

        ! calculate total toroidal current
        allocate(i_tor_int(size(i_tor)))
        ierr = calc_int(i_tor*norm_transf(:,1),&
            &flux_t_h(:,0)/flux_t_h(n_r_eq,0),i_tor_int)
        if (use_normalization) i_tor_int = i_tor_int * r_0*b_0/mu_0_original
        chckerr('')

        ! user output
        call writo('Initialize boundary')
        call lvl_ud(1)

        ! full 0..2pi boundary
        if (ias.eq.0) then                                          ! symmetric (so nchi is odd)
            n_b = nchi*2-2                                          ! -2 because of overlapping points at 0, pi
        else                                                        ! asymmetric (so nchi is aumented by one)
            n_b  = nchi-1                                           ! -1 because of overlapping points at 0
        end if
        allocate(bh_0(n_b,2))                                       ! HELENA coords (no overlap at 0)
        allocate(theta_b(n_b))                                      ! geometrical poloidal angle at boundary (overlap at 0)
        if (ias.eq.0) then                                          ! symmetric
            bh_0(1:nchi,1) = r_h_loc(:,n_r_eq)
            bh_0(1:nchi,2) = z_h_loc(:,n_r_eq)
            do id = 1,nchi-2
                bh_0(nchi+id,1) = r_h_loc(nchi-id,n_r_eq)
                bh_0(nchi+id,2) = -z_h_loc(nchi-id,n_r_eq)
            end do
        else
```

```
            bh_0(:,1) = r_h_loc(1:n_b,n_r_eq)
            bh_0(:,2) = z_h_loc(1:n_b,n_r_eq)
        end if
        rz_b_0(1) = sum(r_h_loc(:,1))/size(r_h_loc,1)
        rz_b_0(2) = sum(z_h_loc(:,1))/size(z_h_loc,1)
        theta_b = atan2(bh_0(:,2)-rz_b_0(2),bh_0(:,1)-rz_b_0(1))
        where (theta_b.lt.0) theta_b = theta_b + 2*pi
        call writo('Magnetic axis used for geometrical coordinates:')
        call lvl_ud(1)
        call writo('('//trim(r2str(rz_b_0(1)))//'m, '//&
            &trim(r2str(rz_b_0(2)))//'m)')
        call lvl_ud(-1)

#if ldebug
        if (ltest) then
            call writo('Test with circular tokamak?')
            call lvl_ud(1)
            if(get_log(.false.)) then
                call writo('Testing with circular tokamak:')
                call lvl_ud(1)
                call writo('R = 3/2 + 1/2 cos(θ)')
                call writo('Z =       1/2 sin(θ)')
                call writo('with θ equidistant')
                call lvl_ud(-1)
                theta_b = [((id-1._dp)/n_b*2*pi,id=1,n_b)]
                bh_0(:,1) = 1.5_dp + 0.5*cos(theta_b)
                bh_0(:,2) = 0.5*sin(theta_b)
            end if
            call lvl_ud(-1)
        end if
#endif

        ! plot with HDF5
        allocate(theta_geo(grid_eq%n(1),1,grid_eq%loc_n_r))
        do kd = 1,grid_eq%loc_n_r
            theta_geo(:,1,kd) = atan2(z_h_loc(:,kd)-rz_b_0(2),&
                &r_h_loc(:,kd)-rz_b_0(1))
        end do
        where (theta_geo.lt.0._dp) theta_geo = theta_geo + 2*pi
        allocate(xyz_plot(grid_eq%n(1),1,grid_eq%loc_n_r,3))
        ierr = calc_xyz_grid(grid_eq,grid_eq,xyz_plot(:,:,:,1),&
            &xyz_plot(:,:,:,2),xyz_plot(:,:,:,3))
        chckerr('')
        call plot_hdf5('theta_geo','theta_geo',theta_geo,&
            &tot_dim=[grid_eq%n(1),1,grid_eq%n(3),3],&
            &loc_offset=[0,0,grid_eq%i_min-1,0],&
            &x=xyz_plot(:,:,:,1),y=xyz_plot(:,:,:,2),z=xyz_plot(:,:,:,3),&
            &descr='geometric poloidal angle used to create the VMEC &
            &input file')
        deallocate(xyz_plot)

        ! plot R and Z
        plot_name(1) = 'RZ'
        plot_title = ['R_H','Z_H']
        call print_ex_2d(plot_title(1:2),plot_name(1),bh_0,&
            &x=reshape(theta_b,[n_b,1])/pi,&
            &draw=.false.)
        call draw_ex(plot_title(1:2),plot_name(1),2,1,.false.)

        ! set up auxiliary variable
        flux_name = ['poloidal','toroidal']

        call lvl_ud(-1)

        ! plot properties
        plot_dim = [100,20]
        plot_lims(:,1) = [0.0_dp,2.0_dp]
        plot_lims(:,2) = [0.5_dp,2.0_dp]
        call writo('Change plot properties from defaults?')
        call lvl_ud(1)
        do id = 1,2
            call writo(trim(i2str(plot_dim(id)))//' geometrical '//&
                &flux_name(id)//' points on range '//&
                &trim(r2strt(plot_lims(1,id)))//' pi .. '//&
                &trim(r2strt(plot_lims(2,id)))//' pi')
        end do
        call lvl_ud(-1)
        if (get_log(.false.)) then
            call lvl_ud(1)
            do id = 1,2
                call writo('number of '//flux_name(id)//' points?')
                plot_dim(id) = get_int(lim_lo=2)
                call writo('lower limit of '//flux_name(id)//&
                    &' range [pi]?')
                plot_lims(1,id) = get_real()
                call writo('upper limit of '//flux_name(id)//&
                    &' range [pi]?')
```

```
                plot_lims(2,id) = get_real()
            end do
            call lvl_ud(-1)
        end if

        ! get ncurr
        call writo('VMEC current style?')
        call lvl_ud(1)
        call writo('0: prescribe iota')
        call writo('1: prescribe toroidal current')
        ncurr = get_int(lim_lo=0,lim_hi=1)
        call lvl_ud(-1)

        ! get input about equilibrium perturbation
        ! Note: negative  M values will  be converted  to negative N  values and
        ! Possibly negative delta's at the end.
        call writo('Do you want to perturb the equilibrium?')
        pert_eq = get_log(.false.)

        zero_n_pert = .false.
        if (pert_eq) then
            ! get perturbation style
            call writo('Which perturbation do you want to describe?')
            call lvl_ud(1)
            call writo('1: plasma boundary position')
            call writo('2: B_tor magnetic ripple with fixed N')
            call lvl_ud(-1)
            pert_style = get_int(lim_lo=1,lim_hi=2)

            ! for style 2, get proportionality file
            if (pert_style.eq.2) then
                ! find file
                found = .false.
                do while (.not.found)
                    call writo('Proportionality file name?')
                    call lvl_ud(1)
                    read(*,*,iostat=ierr) prop_b_tor_file_name
                    err_msg = 'failed to read prop_B_tor_file_name'
                    chckerr(err_msg)

                    ! open
                    do kd = 0,2
                        call writo('Trying "'//path_prefix(1:kd*3)//&
                            &trim(prop_b_tor_file_name)//'"')
                        open(prop_b_tor_i,file=path_prefix(1:kd*3)//&
                            &trim(prop_b_tor_file_name),iostat=ierr,&
                            &status='old')
                        if (ierr.eq.0) then
                            call lvl_ud(1)
                            call writo("Success")
                            call lvl_ud(-1)
                            exit
                        end if
                    end do
                    if (ierr.eq.0) then
                        found = .true.
                    else
                        call writo('Could not open file "'//&
                            &trim(prop_b_tor_file_name)//'"')
                    end if

                    call lvl_ud(-1)
                end do

                ! read file
                call writo('Analyzing perturbation proportionality file "'&
                    &//trim(prop_b_tor_file_name)//'"')
                call lvl_ud(1)
                n_prop_b_tor = count_lines(prop_b_tor_i)
                allocate(prop_b_tor(n_prop_b_tor,2))
                ierr = skip_comment(prop_b_tor_i,&
                    &file_name=prop_b_tor_file_name)
                chckerr('')
                do id = 1,n_prop_b_tor
                    read(prop_b_tor_i,*,iostat=ierr) prop_b_tor(id,:)
                end do

                ! user info
                call writo(trim(i2str(n_prop_b_tor))//&
                    &' poloidal points '//&
                    &trim(r2strt(minval(prop_b_tor(:,1))))//'..'//&
                    &trim(r2strt(maxval(prop_b_tor(:,1)))))
                call writo('The proportionality factor should be tabulated &
                    &in a geometrical angle that has the SAME origin as &
                    &the one used here',alert=.true.)

                ! interpolate the proportionality  factor on the geometrical
```

```
                ! poloidal angle used  here (as opposed to the  one in which
                ! it was tabulated)
                ierr = order_per_fun(prop_b_tor,prop_b_tor_ord,1,&              ! overlap of 1 is enough for splines
                    &tol=0.5_dp*pi/n_prop_b_tor)                                ! set appropriate tolerance: a quarter of a equidistant grid
                chckerr('')
                deallocate(prop_b_tor)
                allocate(prop_b_tor_interp(n_b))
                ierr = spline(prop_b_tor_ord(:,1),prop_b_tor_ord(:,2),&
                    &theta_b,prop_b_tor_interp,ord=norm_disc_prec_eq,&
                    &bcs=[-1,-1])
                chckerr('')
                deallocate(prop_b_tor_ord)
                if (grid_eq%n(2).ne.1) call writo('There should be only 1 &
                    &geodesical position, but there are '//&
                    &trim(i2str(grid_eq%n(2))),warning=.true.)

                ! smooth prop_B_tor_interp
                prop_b_tor_smooth = 1._dp
                call writo('Do you want to smooth the perturbation?')
                if (get_log(.false.)) prop_b_tor_smooth = &
                    &get_real(lim_lo=0._dp,lim_hi=1._dp)

                ! calculate NUFFT
                ierr = nufft(theta_b,prop_b_tor_interp,prop_b_tor_interp_f)
                chckerr('')
                deallocate(prop_b_tor_interp)

                call lvl_ud(-1)
            end if

            ! get perturbation type
            call writo('How do you want to prescribe the perturbation?')
            call lvl_ud(1)
            call writo('1: through a file with Fourier modes in &
                &geometrical coordinates')
            call lvl_ud(1)
            call writo('It should provide N M delta_cos delta_sin in four &
                &columns')
            call writo('Rows starting with "#" are ignored')
            call lvl_ud(-1)
            call writo('2: same as 1 but interactively')
            call writo('3: through a 2-D map in R, Z for constant &
                &geometrical coordinates')
            call lvl_ud(1)
            call writo('It should provide R, Z and delta')
            call writo('Rows starting with "#" are ignored')
            call lvl_ud(-1)
            pert_type = get_int(lim_lo=1,lim_hi=3)
            call lvl_ud(-1)

            select case (pert_type)
                case (1,3)                                                      ! files
                    found = .false.
                    do while (.not.found)
                        ! user input
                        call writo('File name?')
                        call lvl_ud(1)
                        read(*,*,iostat=ierr) hel_pert_file_name
                        err_msg = 'failed to read HEL_pert_file_name'
                        chckerr(err_msg)

                        ! open
                        do kd = 0,2
                            call writo('Trying "'//path_prefix(1:kd*3)//&
                                &trim(hel_pert_file_name)//'"')
                            open(hel_pert_i,file=path_prefix(1:kd*3)//&
                                &trim(hel_pert_file_name),iostat=ierr,&
                                &status='old')
                            if (ierr.eq.0) then
                                call lvl_ud(1)
                                call writo("Success")
                                call lvl_ud(-1)
                                exit
                            end if
                        end do
                        if (ierr.eq.0) then
                            found = .true.
                        else
                            call writo('Could not open file "'//&
                                &trim(hel_pert_file_name)//'"')
                        end if

                        call lvl_ud(-1)
                    end do

                    call writo('Parsing "'//trim(hel_pert_file_name)//'"')
```

```
                    call lvl_ud(1)

                    if (pert_type.eq.1) then                                    ! Fourier modes in geometrical coordinates
                        ! get number of lines
                        tot_nr_pert = count_lines(hel_pert_i)
                    else if (pert_type.eq.3) then                               ! 2-D map of perturbation
                        ! ask for vertical shift
                        call writo('Was the equilibrium shifted &
                            &vertically?')
                        if (abs(eq_vert_shift).gt.0._dp) then
                            call lvl_ud(1)
                            call writo('Note: If you already shifted just &
                                &now, don''t do it again!',alert=.true.)
                            call lvl_ud(-1)
                        end if
                        pert_map_vert_shift = 0._dp
                        if (get_log(.false.)) then
                            call writo('How much higher [m] should the &
                                &equilibrium be in the real world?')
                            pert_map_vert_shift = get_real()
                            call writo('The perturbation map will be &
                                &shifted by '//&
                                &trim(r2strt(-pert_map_vert_shift))//&
                                &'m to compensate for this')
                        end if

                        ! get map
                        ierr = skip_comment(hel_pert_i,&
                            &file_name=hel_pert_file_name)
                        chckerr('')
                        read(hel_pert_i,*) n_pert_map(1)                        ! nr. of points in R
                        allocate(pert_map_r(n_pert_map(1)))
                        do kd = 1,n_pert_map(1)
                            read(hel_pert_i,*) pert_map_r(kd)
                        end do
                        ierr = skip_comment(hel_pert_i,&
                            &file_name=hel_pert_file_name)
                        chckerr('')
                        read(hel_pert_i,*) n_pert_map(2)                        ! nr. of points in R
                        allocate(pert_map_z(n_pert_map(2)))
                        do kd = 1,n_pert_map(2)
                            read(hel_pert_i,*) pert_map_z(kd)
                        end do
                        pert_map_z = pert_map_z - pert_map_vert_shift
                        ierr = skip_comment(hel_pert_i,&
                            &file_name=hel_pert_file_name)
                        chckerr('')
                        allocate(pert_map(n_pert_map(1),n_pert_map(2)))
                        do kd = 1,n_pert_map(1)
                            read(hel_pert_i,*) pert_map(kd,:)
                        end do

                        ! plot map
                        allocate(r_plot(n_pert_map(1),n_pert_map(2),1))
                        allocate(z_plot(n_pert_map(1),n_pert_map(2),1))
                        do kd = 1,n_pert_map(2)
                            r_plot(:,kd,1) = pert_map_r
                        end do
                        do kd = 1,n_pert_map(1)
                            z_plot(kd,:,1) = pert_map_z
                        end do
                        call plot_hdf5('pert_map','pert_map',&
                            &reshape(pert_map,[n_pert_map,1]),&
                            &x=r_plot,y=z_plot,descr=&
                            &'perturbation map for '//&
                            &trim(hel_pert_file_name))
                        deallocate(r_plot,z_plot)

                        ! test
                        if (minval(r_h_loc).lt.minval(pert_map_r) .or. &
                            &maxval(r_h_loc).gt.maxval(pert_map_r) .or. &
                            &minval(z_h_loc).lt.minval(pert_map_z) .or. &
                            &maxval(z_h_loc).gt.maxval(pert_map_z)) then
                            ierr = 1
                            call writo('Are you sure you have specified &
                                &a normalization factor R_0?',alert=.true.)
                            err_msg = 'R and Z are not contained in &
                                &perturbation map'
                            chckerr(err_msg)
                        end if

                        ! set up 2-D spline
                        allocate(pert_map_interp(n_b))
                        bcs(:,1) = [0,0]                                        ! not a knot
                        bcs(:,2) = [0,0]                                        ! not a knot
                        call ezspline_init(f_spl,n_pert_map(1),n_pert_map(2),&
                            &bcs(:,1),bcs(:,2),ierr)
```

```
                        call ezspline_error(ierr)
                        chckerr('')

                        ! set grid
                        f_spl%x1 = pert_map_r
                        f_spl%x2 = pert_map_z

                        ! set up
                        call ezspline_setup(f_spl,pert_map,ierr,&
                            &exact_dim=.true.)                              ! match exact dimensions, none of them old Fortran bullsh*t!
                        call ezspline_error(ierr)
                        chckerr('')

                        ! interpolate
                        do id = 1,n_b
                            call ezspline_interp(f_spl,bh_0(id,1),bh_0(id,2),&
                                &pert_map_interp(id),ierr)
                            call ezspline_error(ierr)
                            chckerr('')
                        end do
                        call print_ex_2d('ripple','pert_map_interp',&
                            &pert_map_interp,x=theta_b,draw=.false.)
                        call draw_ex(['ripple'],'pert_map_interp',&
                            &1,1,.false.)

                        ! calculate NUFFT
                        ierr = nufft(theta_b,pert_map_interp,&
                            &pert_map_interp_f)
                        chckerr('')
                        tot_nr_pert = size(pert_map_interp_f,1)*2           ! need both positive and negative n

                        ! get toroidal mode number
                        call writo('toiroidal period for 2-D map?')
                        pert_map_n_loc = get_int(lim_lo=0)
                    end if

                    call lvl_ud(-1)
                case (2)                                                    ! interactively
                    ! user input
                    call writo('Prescribe interactively')
                    call writo('How many different combinations of &
                        &toroidal and poloidal mode numbers (N,M)?')
                    tot_nr_pert = get_int(lim_lo=1)
            end select

            ! set up n, m and delta
            allocate(n_pert(tot_nr_pert+1))
            allocate(delta(tot_nr_pert+1,0:0,2))                            ! start with only M = 0
            delta = 0._dp
            nr_n = 0
            do jd = 1,tot_nr_pert
                select case (pert_type)
                    case (1)                                                ! file with Fourier modes in geometrical coordinates
                        ierr = skip_comment(hel_pert_i,&
                            &file_name=hel_pert_file_name)
                        chckerr('')
                        read(hel_pert_i,*,iostat=ierr) n_loc, m_loc, &
                            &delta_loc
                        err_msg = 'Could not read file '//&
                            &trim(hel_pert_file_name)
                        chckerr(err_msg)
                    case (2)                                                ! same as 1 but interactively
                        call writo('For mode '//trim(i2str(jd))//'/'//&
                            &trim(i2str(tot_nr_pert))//':')
                        call lvl_ud(1)

                        ! input
                        call writo('Toroidal mode number N?')
                        n_loc = get_int()
                        call writo('Poloidal mode number M?')
                        m_loc = get_int()
                        call writo('Perturbation strength ~ cos('//&
                            &trim(i2str(m_loc))//' θ - '//&
                            &trim(i2str(n_loc))//' ζ)?')
                        delta_loc(1) = get_real()
                        call writo('Perturbation strength ~ sin('//&
                            &trim(i2str(m_loc))//' θ - '//&
                            &trim(i2str(n_loc))//' ζ)?')
                        delta_loc(2) = get_real()

                        call lvl_ud(-1)
                    case (3)                                                ! file with Fourier modes in general coordinates
                        m_loc = (jd-1)/2                                    ! so the output is 0,0,1,1,2,2,3,3,4,4,...
                        delta_loc = pert_map_interp_f(m_loc+1,:)*0.5_dp     ! both + and - helicity, so need to divide
```

```
amplitude by 2
                        n_loc = pert_map_n_loc
                        if (mod(jd,2).eq.0) n_loc = -n_loc
                end select

                ! has N = 0 been prescribed?
                if (n_loc.eq.0) zero_n_pert = .true.

                ! bundle into n_pert
                n_id = nr_n+1                                           ! default new N in next index
                do id = 1,nr_n
                    if (n_loc.eq.n_pert(id)) n_id = id
                end do
                if (n_id.gt.nr_n) then                                  ! new N
                    nr_n = n_id                                         ! increment number of N
                    n_pert(n_id) = n_loc                                ! with value n_loc
                end if
                m_range = [lbound(delta,2),ubound(delta,2)]
                if (m_loc.gt.m_range(1) .or. m_loc.lt.m_range(2)) then  ! enlarge delta
                    allocate(delta_copy(tot_nr_pert+1,&
                        &m_range(1):m_range(2),2))
                    delta_copy = delta
                    deallocate(delta)
                    allocate(delta(tot_nr_pert+1,&
                        &min(m_range(1),m_loc):max(m_range(2),m_loc),2))
                    delta = 0._dp
                    delta(:,m_range(1):m_range(2),:) = delta_copy
                    deallocate(delta_copy)
                end if
                delta(n_id,m_loc,:) = delta(n_id,m_loc,:) + delta_loc   ! and perturbation amplitude for cos and sin
with value delta_loc

#if ldebug
                if (debug_create_vmec_input) then
                    call writo('perturbation '//trim(i2str(jd))//&
                        &' at position '//trim(i2str(n_id))//', adding ('//&
                        &trim(r2strt(delta(n_id,m_loc,1)))//', '//&
                        &trim(r2strt(delta(n_id,m_loc,2)))//') at (n,m) = ('//&
                        &trim(i2str(n_loc))//', '//trim(i2str(m_loc))//')')
                    call lvl_ud(1)
                    do kd = lbound(delta,2),ubound(delta,2)
                        call writo('delta ('//trim(i2str(kd))//') = ('//&
                            &trim(r2strt(delta(n_id,kd,1)))//', '//&
                            &trim(r2strt(delta(n_id,kd,2)))//')')
                    end do
                    call lvl_ud(-1)
                end if
#endif
            end do

            if (pert_type.eq.1 .or. pert_type.eq.3) close(hel_pert_i)

            ! ask for global rescaling
            ! Note: This  is only valid if R(θ=0) is  the place with the
            ! maximum perturbation
            max_pert_on_axis = sum(sqrt(sum(delta**2,3)))
            select case (pert_style)
                case (1)                                                ! plasma boundary position
                    call writo('Maximum absolute perturbation on axis: '//&
                        &trim(r2strt(100*max_pert_on_axis))//'cm')
                    call writo('Rescale to some value [cm]?')
                case (2)                                                ! B_tor magnetic ripple with fixed N
                    call writo('Maximum relative perturbation on axis: '//&
                        &trim(r2strt(100*max_pert_on_axis))//'%')
                    call writo('Rescale to some value [%]?')
            end select
            mult_fac = max_pert_on_axis
            if (get_log(.false.)) then
                mult_fac = get_real()
                mult_fac = mult_fac * 0.01_dp
            end if
            delta = delta * mult_fac / max_pert_on_axis

            ! add zero to the peturbations if it's not yet there
            ! (for the bubble sort)
            if (.not.zero_n_pert) then
                nr_n = nr_n+1
                n_pert(nr_n) = 0
            end if
        else
            nr_n = 1
        end if

        ! user output
        call writo('Set up axisymmetric data')
        call lvl_ud(1)

```

```
        ! calculate output for unperturbed R and Z
        nr_m_max = (n_b-1)/2                                                    ! maximum perturbation
        if (pert_eq) then
            nr_m_max = max(nr_m_max, max(-lbound(delta,2),ubound(delta,2)))     ! correct if less than delta
            if (pert_style.eq.2) &
                &nr_m_max = max(nr_m_max, size(prop_b_tor_interp_f,1)-1)        ! correct if less than proportionality map
        end if
        plot_name(1) = 'R_F'
        plot_name(2) = 'Z_F'
        allocate(b_f(nr_n,-nr_m_max:nr_m_max,2,2))                              ! (tor modes, pol modes, cos/sin (m θ), R/Z)
        ! Note: B_F will later be reallocated to run from 0:nr_m_max in index 2
        b_f = 0._dp
        do kd = 1,2
            ! user output
            call writo('analyzing '//trim(plot_name(kd)))
            call lvl_ud(1)

            ! NUFFT
            ierr = nufft(theta_b,bh_0(:,kd),b_f_dum,plot_name(kd))
            chckerr('')
            b_f(1,0:size(b_f_dum,1)-1,:,kd) = b_f_dum

#if ldebug
            if (debug_create_vmec_input) then
                allocate(bh_0_alt(size(bh_0,1),size(b_f_dum,1)))

                call writo('Comparing R or Z with reconstruction &
                    &through Fourier coefficients')
                bh_0_alt(:,1) = b_f_dum(1,1)
                do id = 1,size(b_f_dum,1)-1
                    bh_0_alt(:,id+1) = bh_0_alt(:,id) + &
                        &b_f_dum(id+1,1)*cos(id*theta_b) + &
                        &b_f_dum(id+1,2)*sin(id*theta_b)
                end do
                call print_ex_2d(['orig BH','alt BH '],'',&
                    &reshape([bh_0(:,kd),bh_0_alt(:,size(b_f_dum,1))],&
                    &[size(bh_0,1),2]),x=&
                    &reshape([theta_b],[size(bh_0,1),1]))

                call writo('Plotting Fourier approximation')
                call print_ex_2d(['alt BH'],'TEST_'//trim(plot_name(kd))//&
                    &'_F_series',bh_0_alt,&
                    &x=reshape([theta_b],[size(bh_0,1),1]))
                call draw_ex(['alt BH'],'TEST_'//trim(plot_name(kd))//&
                    &'_F_series',size(b_f_dum,1),1,.false.)

                call writo('Making animation')
                call print_ex_2d(['alt BH'],'TEST_'//trim(plot_name(kd))//&
                    &'_F_series_anim',bh_0_alt,&
                    &x=reshape([theta_b],[size(bh_0,1),1]))
                call draw_ex(['alt BH'],'TEST_'//trim(plot_name(kd))//&
                    &'_F_series_anim',size(b_f_dum,1),1,.false.,&
                    &is_animated=.true.)

                deallocate(bh_0_alt)
            end if
#endif
            deallocate(b_f_dum)

            call lvl_ud(-1)
        end do

        ! plot axisymetric boundary in 3D
        ! Note: Still using -nr_m_max:nr_m_max for poloidal modes
        call plot_boundary(b_f(1:1,:,:,:),[-nr_m_max,nr_m_max],[0],&
            &'last_fs',plot_dim,plot_lims)

        call lvl_ud(-1)

        ! sort and output
        if (pert_eq) then
            ! user output
            call writo('Set up perturbed data')
            call lvl_ud(1)

            ! sort
            allocate(n_pert_copy(nr_n))
            allocate(delta_copy(nr_n,-nr_m_max:nr_m_max,2))
            delta_copy = 0.0_dp
            n_pert_copy = n_pert(1:nr_n)
            delta_copy(:,lbound(delta,2):ubound(delta,2),:) = delta(1:nr_n,:,:)
            deallocate(n_pert); allocate(n_pert(nr_n)); n_pert = n_pert_copy
            deallocate(delta); allocate(delta(nr_n,-nr_m_max:nr_m_max,2))
            deallocate(n_pert_copy)
            allocate(piv(nr_n))
            call bubble_sort(n_pert,piv)                                        ! sort n
            do jd = 1,nr_n
```

```
            delta(jd,:,:) = delta_copy(piv(jd),:,:)
        end do
        deallocate(piv)

        ! update the index of N = 0 in B_F
        id_n_0 = minloc(abs(n_pert),1)
        if (id_n_0.gt.1) then
            b_f(id_n_0,:,:,:) = b_f(1,:,:,:)
            b_f(1,:,:,:) = 0._dp
        end if

        ! output
        call writo('Summary of perturbation form:')
        call lvl_ud(1)
        do jd = 1,nr_n
            do id = -nr_m_max,nr_m_max
                if (maxval(abs(delta(jd,id,:))).lt.tol_zero) cycle          ! this mode has no amplitude
                if (n_pert(jd).ge.0) then
                    pm(1) = '-'
                else
                    pm(1) = '+'
                end if
                if (delta(jd,id,1).ge.0) then
                    pm(2) = '+'
                else
                    pm(2) = '-'
                end if
                if (jd.eq.1 .and. id.eq.1) pm(2) = ''
                if (delta(jd,id,2).ge.0) then
                    pm(3) = '+'
                else
                    pm(3) = '-'
                end if
                call writo(&
                    &pm(2)//' '//trim(r2strt(abs(delta(jd,id,1))))//&
                    &' cos('//trim(i2str(id))//' θ '//pm(1)//' '//&
                    &trim(i2str(abs(n_pert(jd))))//' ζ) '//&
                    &pm(3)//' '//trim(r2strt(abs(delta(jd,id,2))))//&
                    &' sin('//trim(i2str(id))//' θ '//pm(1)//' '//&
                    &trim(i2str(abs(n_pert(jd))))//' ζ)')
            end do
        end do
        call lvl_ud(-1)

        ! loop  over all  toroidal  modes N  and  include the  corresponding
        ! perturbation, consisting of terms of the form:
        ! δc cos(M θ - N ζ) + δs sin(M θ - N ζ) =
        !   δc [cos(M θ)cos(N ζ) + sin(M θ) sin(N ζ) ] +
        !   δs [sin(M θ)cos(N ζ) - cos(M θ) sin(N ζ) ].
        ! First, however, the δc and δs have to be translated from
        ! an absolute modification  of the radius r = sqrt(R^2  + Z^2) to an
        ! absolute modification of R and Z:
        !    ΔR = δ(m) cos(m θ)
        !    ΔZ = δ(m) sin(m θ),
        ! which simply shifts up and down the index m by one.
        ! Additionally, for perturbation type 2  (2D map), there is an extra
        ! shift before this.
        !
        ! These shifts are done separately  for each toroidal mode number N,
        ! so that  the factors cos(θ M)  and sin(θ M) are  shifted and the
        ! resulting corresponding modal amplitudes  become δc' and δs' for
        ! the term proportional to cos(N ζ):
        !    δc cos(M θ) + δs sin(M θ),
        ! and δc" and δs" for the term proportional to sin(N ζ):
        !   -δs cos(M θ) + δc sin(M θ).
        !
        ! Finally recombining  the results into terms  proportional to cos(m
        ! θ - N ζ) and sin(m θ - N ζ) then results in
        !   (δc'+δs")/2 cos(m θ - N ζ) + (δc'-δs")/2 cos(-m θ - N ζ) +
        !   (δs'-δc")/2 sin(m θ - N ζ) - (δs'+δc")/2 sin(-m θ - N ζ),
        ! which  means that the  different toroidal  modes N can  be treated
        ! independently.
        !
        ! Note that if  there is no shifting  at all, δc =  δc' =  δs" and
        ! δs = δs' = -δc" and m = M, which indeed reduces to
        !   δc cos(M θ - N ζ) + δs sin(M θ - N ζ).
        m_range = [-nr_m_max,nr_m_max]
        pert_n: do jd = 1,nr_n
            ! initialize
            allocate(b_f_dum(-nr_m_max:nr_m_max,2))
            allocate(b_f_dum2(-nr_m_max:nr_m_max,2))
            allocate(b_f_dum3(-nr_m_max:nr_m_max,2))

            ! for R, shift due to cos(θ), for Z, shift due to sin(θ)
            do kd = 1,2                                                     ! iterate over R (kd = 1), then Z (kd = 2)
                b_f_dum = 0._dp
                b_f_dum(1,kd) = 1._dp                                       ! cos(θ) for kd = 1, sin(θ) for kd = 2
```

```
                    if (pert_style.eq.2) then                                   ! shift due to proportionality map
                        b_f_dum2(0:ubound(prop_b_tor_interp_f,1),:) = &
                            &prop_b_tor_interp_f
                        call shift_f(m_range,m_range,m_range,&
                            &b_f_dum,b_f_dum2,b_f_dum3)
                        b_f_dum = b_f_dum3
                    end if

                    ! calculate δ' due to δc cos(M θ) + δs sin(M θ)
                    b_f_dum2(:,1) = delta(jd,:,1)                               ! ~ cos
                    b_f_dum2(:,2) = delta(jd,:,2)                               ! ~ sin
                    call shift_f(m_range,m_range,m_range,&
                        &b_f_dum,b_f_dum2,b_f_dum3)
                    b_f_dum3 = b_f_dum3*0.5_dp
                    !  (δc'+δs")/2 cos(m θ - N ζ)
                    b_f(jd,:,1,kd) = b_f(jd,:,1,kd) + b_f_dum3(:,1)
                    !  (δs'-δc")/2 sin(m θ - N ζ)
                    b_f(jd,:,2,kd) = b_f(jd,:,2,kd) + b_f_dum3(:,2)
                    !  (δc'-δs")/2 cos(-m θ - N ζ)
                    b_f(jd,:,1,kd) = b_f(jd,:,1,kd) + &
                        &b_f_dum3(nr_m_max:-nr_m_max:-1,1)
                    ! -(δs'+δc")/2 sin(-m θ - N ζ)
                    b_f(jd,:,2,kd) = b_f(jd,:,2,kd) - &
                        &b_f_dum3(nr_m_max:-nr_m_max:-1,2)

                    ! calculate δ' due to -δs cos(M θ) + δc sin(M θ)
                    b_f_dum2(:,1) = -delta(jd,:,2)                              ! ~ cos
                    b_f_dum2(:,2) = delta(jd,:,1)                               ! ~ sin
                    call shift_f(m_range,m_range,m_range,&
                        &b_f_dum,b_f_dum2,b_f_dum3)
                    b_f_dum3 = b_f_dum3*0.5_dp
                    !  (δc'+δs")/2 cos(m θ - N ζ)
                    b_f(jd,:,1,kd) = b_f(jd,:,1,kd) + b_f_dum3(:,2)
                    !  (δs'-δc")/2 sin(m θ - N ζ)
                    b_f(jd,:,2,kd) = b_f(jd,:,2,kd) - b_f_dum3(:,1)
                    !  (δc'-δs")/2 cos(-m θ - N ζ)
                    b_f(jd,:,1,kd) = b_f(jd,:,1,kd) - &
                        &b_f_dum3(nr_m_max:-nr_m_max:-1,2)
                    ! -(δs'+δc")/2 sin(-m θ - N ζ)
                    b_f(jd,:,2,kd) = b_f(jd,:,2,kd) - &
                        &b_f_dum3(nr_m_max:-nr_m_max:-1,1)
                end do

                deallocate(b_f_dum)
                deallocate(b_f_dum2)
                deallocate(b_f_dum3)
            end do pert_n

            allocate(b_f_copy(2*nr_n,0:nr_m_max,2,2))                           ! swap negative m for inverse n
            allocate(n_pert_copy(2*nr_n))
            b_f_copy = 0._dp
            n_pert_copy = 0
            nr_n = 0                                                            ! start at zero again
            allocate(b_f_dum(0:nr_m_max,2))
            pert_n_convert: do jd = 1,size(b_f,1)
                do kd = 1,2                                                     ! loop over negative and positive ranges
                    ! loop over R and Z
                    do ld = 1,2
                        ! initialize B_F dummy, then set it up
                        b_f_dum = 0._dp
                        if (kd.eq.1) then
                            ! set local n for negative range: invert n and m
                            n_loc = -n_pert(jd)
                            b_f_dum(0:nr_m_max,1) = b_f(jd,0:-nr_m_max:-1,1,ld) ! do not invert cosine factors
                            b_f_dum(0:nr_m_max,2) = -b_f(jd,0:-nr_m_max:-1,2,ld)! invert sine factors
                        else
                            ! set local n for positive range
                            n_loc = n_pert(jd)
                            b_f_dum(0:nr_m_max,:) = b_f(jd,0:nr_m_max,:,ld)
                        end if
                        b_f_dum(0,:) = b_f_dum(0,:)*0.5_dp                      ! positive and negative range share half each

                        if (maxval(abs(b_f_dum)).gt.tol_zero) then
                            ! bundle into n_pert
                            n_id = nr_n+1                                       ! default new N in next index
                            do id = 1,nr_n                                      ! check all previous, though
                                if (n_loc.eq.n_pert_copy(id)) n_id = id
                            end do
                            if (n_id.gt.nr_n) then                              ! new N
                                nr_n = n_id                                     ! increment number of N
                                n_pert_copy(n_id) = n_loc                       ! with value n_loc
                            end if
#if ldebug
                            if (debug_create_vmec_input) then
                                call writo('putting n = '//trim(i2str(n_loc))//&
                                    &' in index '//trim(i2str(n_id))//':')
```

```
                                if (kd.eq.1) then
                                    call writo('(negative range)')
                                else
                                    call writo('(positive range)')
                                end if
                                if (ld.eq.1) then
                                    call writo('(for R)')
                                else
                                    call writo('(for Z)')
                                end if
                                call lvl_ud(1)
                                do id = 0,nr_m_max
                                    if (maxval(abs(b_f_dum(id,:)))&
                                        &.gt.tol_zero) then
                                        call writo('B_F ('//trim(i2str(id))//&
                                            &') = ('//&
                                            &trim(r2strt(b_f_dum(id,1)))//&
                                            &', '//&
                                            &trim(r2strt(b_f_dum(id,2)))//')')
                                    end if
                                end do
                                call lvl_ud(-1)
                            end if
#endif

                            ! update B_F_copy
                            b_f_copy(n_id,:,:,ld) = b_f_copy(n_id,:,:,ld) + &
                                &b_f_dum
                        end if
                    end do
                end do
            end do pert_n_convert

            ! copy back to B_F and n_pert
            deallocate(b_f)
            deallocate(n_pert)
            allocate(b_f(nr_n,0:nr_m_max,2,2))
            allocate(n_pert(nr_n))
            b_f = b_f_copy(1:nr_n,:,:,:)
            n_pert = n_pert_copy(1:nr_n)
            deallocate(b_f_copy)
            deallocate(n_pert_copy)

            ! plot boundary in 3D
            ! Note: Using 0:nr_m_max for poloidal modes
            call plot_boundary(b_f,[0,nr_m_max],n_pert(1:nr_n),'last_fs_pert',&
                &plot_dim,plot_lims)
            call lvl_ud(-1)

            ! set nfp: save greatest common denominator of N into nfp, excluding
            ! N=0
            do jd = 2,nr_n
                if (jd.eq.2) then
                    nfp = n_pert(jd)
                else
                    nfp = gcd(nfp,n_pert(jd))
                end if
            end do
            if (nfp.ne.0) then
                n_pert = n_pert/nfp
            else
                nfp = 1
            end if
        else
            ! migrate negative poloidal modes to positive in B_F_copy
            allocate(b_f_copy(1,0:nr_m_max,2,2))                              ! (tor modes, pol modes, cos/sin (m θ), R/Z)
            do id = 0,nr_m_max
                b_f_copy(1,id,1,:) = b_f(1,id,1,:) + b_f(1,-id,1,:)           ! do not invert cosine factors
                b_f_copy(1,id,2,:) = b_f(1,id,2,:) - b_f(1,-id,2,:)           ! invert sine factors
            end do
            b_f_copy(1,0,:,:) = b_f_copy(1,0,:,:)*0.5_dp                      ! positive and negative range share half each
            deallocate(b_f)
            allocate(b_f(1,0:nr_m_max,2,2))
            b_f = b_f_copy
            deallocate(b_f_copy)

            ! set the index of N = 0 in B_F
            id_n_0 = 1

            ! set nfp to one
            nfp = 1
        end if

        ! Note: From now on the dimensions of B_F are (nr_n,0:nr_m_max,2,2)

        ! user output
```

```
        file_name = "input."//trim(eq_name)
        if (pert_eq) then
            select case (pert_type)
                case (1)                                                    ! Fourier modes in geometrical coordinates
                    file_name = trim(file_name)//'_'//&
                        &trim(hel_pert_file_name)
                case (2)                                                    ! specified manually
                    do jd = 1,nr_n
                        file_name = trim(file_name)//'_N'//&
                            &trim(i2str(n_pert(jd)))
                        do kd = 0,nr_m_max
                            file_name = trim(file_name)//'M'//&
                                &trim(i2str(kd))
                        end do
                    end do
                case (3)                                                    ! 2-D map
                    file_name = trim(file_name)//'_2DMAP'
            end select
        end if
        call writo('Generate VMEC input file "'//trim(file_name)//'"')
        call writo('This can be used for VMEC porting')
        call lvl_ud(1)

        ! detect whether there is stellarator symmetry
        stel_sym = .false.
        if (maxval(abs(b_f(:,:,2,1)))/maxval(abs(b_f(:,:,1,1))) &
            &.lt. m_tol .and. &                                             ! R_s << R_c
            maxval(abs(b_f(:,:,1,2)))/maxval(abs(b_f(:,:,2,2))) &
            &.lt. m_tol) &                                                  ! R_c << Z_s
            &stel_sym = .true.
        if (stel_sym) then
            call writo("The equilibrium configuration has stellarator &
                &symmetry")
        else
            call writo("The equilibrium configuration does not have &
                &stellarator symmetry")
        end if

        ! find out how many poloidal modes would be necessary
        rec_min_m = 1
        norm_b_h = maxval(abs(b_f))
        do id = 0,nr_m_max
            if (maxval(abs(b_f(:,id,:,1)/norm_b_h)).gt.m_tol .or. &         ! for R
                &maxval(abs(b_f(:,id,:,2)/norm_b_h)).gt.m_tol) &            ! for Z
                &rec_min_m = id
        end do
        call writo("Detected recommended number of poloidal modes: "//&
            &trim(i2str(rec_min_m)))

        ! possibly get maximum number of modes to plot
        max_n_b_output = rec_min_m
        call writo('Do you want to change the maximum number of modes &
            &from current '//trim(i2str(max_n_b_output))//'?')
        change_max_n_b_output = get_log(.false.)
        if (change_max_n_b_output) then
            call writo('Maximum number of modes to output?')
            max_n_b_output = get_int(lim_lo=1)
        end if

        ! interpolate for VMEC (V) and Jorek (J) output
        s_vj = [((kd-1._dp)/(size(s_vj)-1),kd=1,size(s_vj))]
        allocate(psi_t(grid_eq%n(3)))
        psi_t = eq_1%flux_t_E(:,0)/eq_1%flux_t_E(grid_eq%n(3),0)
        do id = 0,1
            ierr = spline(psi_t,eq_1%pres_E(:,id),s_vj,pres_vj(:,id),&
                &ord=norm_disc_prec_eq)
            chckerr('')
        end do
        if (use_normalization) then
            pres_vj = pres_vj*pres_0
            pres_vj(:,1) = pres_vj(:,1) / psi_0
        end if
        ierr = spline(psi_t,rbphi_h(:,0),s_vj,f_vj,ord=norm_disc_prec_eq)
        chckerr('')
        if (use_normalization) f_vj = f_vj * r_0*b_0
        ierr = spline(psi_t,ffp,s_vj,ffp_vj,ord=norm_disc_prec_eq)
        chckerr('')
        if (use_normalization) then
            ffp_vj = ffp_vj * (r_0*b_0)**2/psi_0
        end if
        ierr = spline(psi_t,flux_p_h(:,0),s_vj,flux_j,ord=norm_disc_prec_eq)
        chckerr('')
        if (use_normalization) flux_j = flux_j*psi_0
        ierr = spline(psi_t,eq_1%q_saf_E(:,0),s_vj,q_saf_vj,&
            &ord=norm_disc_prec_eq)
        chckerr('')
        ierr = spline(psi_t,-eq_1%rot_t_E(:,0),s_vj,rot_t_v,&
```

```
            &ord=norm_disc_prec_eq)
        chckerr('')
        ierr = spline(psi_t,i_tor*norm_transf(:,1),s_vj,i_tor_v,&
            &ord=norm_disc_prec_eq)
        chckerr('')
        ierr = spline(psi_t,i_tor*norm_transf(:,2),s_vj,i_tor_j,&
            &ord=norm_disc_prec_eq)
        chckerr('')
        if (use_normalization) i_tor_v = i_tor_v * r_0*b_0/mu_0_original
        if (use_normalization) i_tor_j = i_tor_j * r_0*b_0/mu_0_original

        ! output to VMEC input file
        open(hel_export_i,status='replace',file=trim(file_name),&
            &iostat=ierr)
        chckerr('Failed to open file')

        write(hel_export_i,"(A)") "!----- General Parameters -----"
        write(hel_export_i,"(A)") "&INDATA"
        write(hel_export_i,"(A)") "MGRID_FILE = 'NONE',"
        write(hel_export_i,"(A)") "PRECON_TYPE = 'GMRES'"
        write(hel_export_i,"(A)") "PREC2D_THRESHOLD = 1.E-9"
        write(hel_export_i,"(A)") "DELT = 1.00E+00,"
        write(hel_export_i,"(A)") "NS_ARRAY = 19, 39, 79, 159, 319"
        write(hel_export_i,"(A)") "LRFP = F"
        write(hel_export_i,"(A,L1)") "LASYM = ", .not.stel_sym
        write(hel_export_i,"(A)") "LFREEB = F"
        write(hel_export_i,"(A)") "NTOR = "//trim(i2str(nr_n-1))                ! -NTOR .. NTOR
        write(hel_export_i,"(A)") "MPOL = "//trim(i2str(max_n_b_output))        ! 0 .. MPOL-1
        write(hel_export_i,"(A)") "TCON0 = 1"
        write(hel_export_i,"(A)") "FTOL_ARRAY = 1.E-6, 1.E-6, 1.E-8, 1.E-10, &
            &5.000E-18,"
        write(hel_export_i,"(A)") "NITER = 100000,"
        write(hel_export_i,"(A)") "NSTEP = 200,"
        write(hel_export_i,"(A)") "NFP = "//trim(i2str(nfp))
        if (use_normalization) then
            write(hel_export_i,"(A)") "PHIEDGE = "//&
                &trim(r2str(-eq_1%flux_t_E(grid_eq%n(3),0)*psi_0))
        else
            write(hel_export_i,"(A)") "PHIEDGE = "//&
                &trim(r2str(-eq_1%flux_t_E(grid_eq%n(3),0)))
        end if
        write(hel_export_i,"(A)") "CURTOR = "//&
            &trim(r2str(i_tor_int(size(i_tor_int))))

        write(hel_export_i,"(A)") ""
        write(hel_export_i,"(A)") ""

        write(hel_export_i,"(A)") "!----- Pressure Parameters -----"
        write(hel_export_i,"(A)") "PMASS_TYPE = 'cubic_spline'"
        write(hel_export_i,"(A)") "GAMMA =  0.00000000000000E+00"
        write(hel_export_i,"(A)") ""
        write(hel_export_i,"(A)",advance="no") "AM_AUX_S ="
        do kd = 1,size(s_vj)
            write(hel_export_i,"(A1,ES23.16)") " ", s_vj(kd)
        end do
        write(hel_export_i,"(A)") ""
        write(hel_export_i,"(A)",advance="no") "AM_AUX_F ="
        do kd = 1,size(pres_vj,1)
            write(hel_export_i,"(A1,ES23.16)") " ", pres_vj(kd,0)
        end do

        write(hel_export_i,"(A)") ""
        write(hel_export_i,"(A)") ""

        write(hel_export_i,"(A)") &
            &"!----- Current/Iota Parameters -----"
        write(hel_export_i,"(A)") "NCURR = "//trim(i2str(ncurr))
        write(hel_export_i,"(A)") ""

        ! iota (is used if ncur = 0)
        write(hel_export_i,"(A)") "PIOTA_TYPE = 'Cubic_spline'"
        write(hel_export_i,"(A)") ""
        write(hel_export_i,"(A)",advance="no") "AI_AUX_S ="
        do kd = 1,size(s_vj)
            write(hel_export_i,"(A1,ES23.16)") " ", s_vj(kd)
        end do
        write(hel_export_i,"(A)") ""
        write(hel_export_i,"(A)",advance="no") "AI_AUX_F ="
        do kd = 1,size(rot_t_v)
            write(hel_export_i,"(A1,ES23.16)") " ", rot_t_v(kd)
        end do
        write(hel_export_i,"(A)") ""

        ! I_tor (is used if ncur = 1)
        write(hel_export_i,"(A)") "PCURR_TYPE = 'cubic_spline_Ip'"
        write(hel_export_i,"(A)") ""
        write(hel_export_i,"(A)",advance="no") "AC_AUX_S ="
```

```fortran
        do kd = 1,size(s_vj)
            write(hel_export_i,"(A1,ES23.16)") " ", s_vj(kd)
        end do
        write(hel_export_i,"(A)") ""
        write(hel_export_i,"(A)",advance="no") "AC_AUX_F ="
        do kd = 1,size(rot_t_v)
            write(hel_export_i,"(A1,ES23.16)") " ", i_tor_v(kd)
        end do

        write(hel_export_i,"(A)") ""
        write(hel_export_i,"(A)") ""

        write(hel_export_i,"(A)") &
            &"!----- Boundary Shape Parameters -----"
        ierr = print_mode_numbers(hel_export_i,b_f(id_n_0,:,:,:),0,&
            &max_n_b_output)
        chckerr('')
        if (pert_eq) then
            do jd = 1,nr_n
                if (jd.eq.id_n_0) cycle                                    ! skip n = 0 as it is already written
                ierr = print_mode_numbers(hel_export_i,&
                    &b_f(jd,:,:,:),n_pert(jd),max_n_b_output)
                chckerr('')
            end do
        end if
        write(hel_export_i,"(A,ES23.16)") "RAXIS = ", rz_b_0(1)
        write(hel_export_i,"(A,ES23.16)") "ZAXIS = ", rz_b_0(2)
        write(hel_export_i,"(A)") "&END"

        close(hel_export_i)

        ! output to output file for free-boundary coil fitting in JOREK
        file_name = "flux_and_boundary_quantities_"//trim(eq_name)//'.jorek'
        open(hel_export_i,status='replace',file=trim(file_name),&
            &iostat=ierr)
        chckerr('Failed to open file')

        ! The normal derivatives in FFp are transformed for JOREK output.
        !   - The difference between the HELENA coordinate system (used in the E
        !   quantities here), and  the JOREK coordinate system is  the fact that
        !   HELENA uses  psi_pol/2pi as  the norma  coordinate while  JOREK uses
        !   psi_pol/psi_pol(s). This step therefore introduces a factor
        !       d/dpsi_J = psi_pol(s)/(2*pi)  d/dpsi_H
        ! However, JOREK  then asks for  a quantity  that has this  exact factor
        ! multiplied back into  it, so that the end results  does not require to
        ! be transformed back. The following lines are therefore commented out.
        !!FFp_VJ = FFp_VJ * max_flux_E/(2*pi)
        !!if (use_normalization) FFp_VJ = FFp_VJ * psi_0                        ! to scale max_flux_E
        if (use_normalization) then
            norm_j = [r_0*rmtog_h,rz_b_0(2),r_0*rmtog_h*b_0*bmtog_h]
        else
            norm_j = [1._dp,1._dp,1._dp]
        end if
        write(hel_export_i,"('! ',A)") 'normalization factors:'
        write(hel_export_i,"('R_geo = ',ES23.16,' ! [m]')") norm_j(1)
        write(hel_export_i,"('Z_geo = ',ES23.16,' ! [m]')") norm_j(2)
        write(hel_export_i,"('F0 = ',ES23.16,' ! [Tm]')") norm_j(3)
        write(hel_export_i,*) ''
        write(hel_export_i,"('! ',7(A23,' '))") 'pol. flux [Tm^2]', &
            &'normalized pol. flux []', 'mu_0 pressure [T^2]', 'F [Tm]', &
            &'-FFp [T^2 m^2/(Wb/rad)]', 'safety factor []', 'I_tor [A]'
        do kd = 1,size(flux_j)
            write(hel_export_i,"('  ',7(ES23.16,' '))") flux_j(kd), &
                &flux_j(kd)/flux_j(size(flux_j)), mu_0_original*pres_vj(kd,0), &
                &f_vj(kd), -ffp_vj(kd), q_saf_vj(kd), i_tor_j(kd)
        end do
        write(hel_export_i,*) ''
        write(hel_export_i,*) '! R [m] and Z [m] of boundary at psi [ ]'
        do kd = 1,n_b
            write(hel_export_i,"('R_boundary(',I4,') = ',ES23.16,&
                &', Z_boundary(',I4,') = ',ES23.16,&
                &', psi_boundary(',I4,') = ',ES23.16)") &
                &kd, bh_0(kd,1), kd, bh_0(kd,2), kd, 1._dp
        end do

        close(hel_export_i)

        ! output different files
        file_name = "jorek_ffprime"
        open(hel_export_i,status='replace',file=trim(file_name),&
            &iostat=ierr)
        chckerr('Failed to open file')
        !write(HEL_export_i,"('! ',2(A23,' '))") 'psi_norm [ ]', '-FFp [T]'
        do kd = 1,size(flux_j)
            write(hel_export_i,"('  ',2(ES23.16,' '))") &
                &flux_j(kd)/flux_j(size(flux_j)), -ffp_vj(kd)
        end do
```

```
        close(hel_export_i)
        file_name = "jorek_temperature"
        open(hel_export_i,status='replace',file=trim(file_name),&
            &iostat=ierr)
        chckerr('Failed to open file')
        !write(HEL_export_i,"('! ',2(A23,' '))") 'psi_norm [ ]', &
            !&'mu_0 pressure [T^2]'
        do kd = 1,size(flux_j)
            write(hel_export_i,"('  ',2(ES23.16,' '))") &
                &flux_j(kd)/flux_j(size(flux_j)), mu_0_original*pres_vj(kd,0)
        end do
        close(hel_export_i)

        ! user output
        call lvl_ud(-1)

        call writo('Done')

        ! wait
        call pause_prog
    contains
        ! plots the boundary of a toroidal configuration
        !> \private
        subroutine plot_boundary(B,m_lims,n,plot_name,plot_dim,plot_lims)
            ! input / output
            real(dp), intent(in) :: b(1:,0:,1:,1:)                          ! cosine and sine of fourier series (tor
modes, pol modes, cos/sin (m theta_geo), R/Z)
            integer, intent(in) :: m_lims(2)                                ! limits of poloidal mode numbers
            integer, intent(in) :: n(:)                                     ! toroidal mode numbers
            character(len=*), intent(in) :: plot_name                       ! name of plot
            integer, intent(in) :: plot_dim(2)                              ! plot dimensions in geometrical angle
            real(dp), intent(in) :: plot_lims(2,2)                          ! limits of plot dimensions [pi]

            ! local variables
            real(dp), allocatable :: ang_plot(:,:,:)                        ! angles of plot (theta,zeta)
            real(dp), allocatable :: xyz_plot(:,:,:)                        ! coordinates of boundary (X,Y,Z)
            integer :: kd, id                                               ! counters
            integer :: m                                                    ! local poloidal mode number

            ! set variables
            allocate(ang_plot(plot_dim(1),plot_dim(2),2))                   ! theta and zeta (geometrical)
            allocate(xyz_plot(plot_dim(1),plot_dim(2),3))                   ! X, Y and Z
            xyz_plot = 0._dp

            ! create grid
            do id = 1,plot_dim(1)
                ang_plot(id,:,1) = pi * (plot_lims(1,1) + (id-1._dp)/&
                    &(plot_dim(1)-1)*(plot_lims(2,1)-plot_lims(1,1)))
            end do
            do kd = 1,plot_dim(2)
                ang_plot(:,kd,2) = pi * (plot_lims(1,2) + (kd-1._dp)/&
                    &(plot_dim(2)-1)*(plot_lims(2,2)-plot_lims(1,2)))
            end do

            ! inverse Fourier transform
            do id = 1,size(n)                                               ! toroidal modes
                do kd = 0,m_lims(2)-m_lims(1)                               ! poloidal modes
                    m = m_lims(1)+kd

                    ! R
                    xyz_plot(:,:,1) = xyz_plot(:,:,1) + &
                        &b(id,kd,1,1)*cos(m*ang_plot(:,:,1)-&
                        &n(id)*ang_plot(:,:,2)) + &
                        &b(id,kd,2,1)*sin(m*ang_plot(:,:,1)-&
                        &n(id)*ang_plot(:,:,2))
                    ! Z
                    xyz_plot(:,:,3) = xyz_plot(:,:,3) + &
                        &b(id,kd,1,2)*cos(m*ang_plot(:,:,1)-&
                        &n(id)*ang_plot(:,:,2)) + &
                        &b(id,kd,2,2)*sin(m*ang_plot(:,:,1)-&
                        &n(id)*ang_plot(:,:,2))
                end do
            end do

            ! print cross-section (R,Z)
            call print_ex_2d(['cross_section_'//trim(plot_name)],&
                &'cross_section_'//trim(plot_name),xyz_plot(:,:,3),&
                &x=xyz_plot(:,:,1),draw=.false.)
            call draw_ex(['cross_section_'//trim(plot_name)],&
                &'cross_section_'//trim(plot_name),plot_dim(2),1,.false.)
            call print_ex_2d(['cross_section_'//trim(plot_name)//'_inv'],&
                &'cross_section_'//trim(plot_name)//'_inv',&
                &transpose(xyz_plot(:,:,3)),x=transpose(xyz_plot(:,:,1)),&
                &draw=.false.)
            call draw_ex(['cross_section_'//trim(plot_name)//'_inv'],&
                &'cross_section_'//trim(plot_name)//'_inv',plot_dim(1),1,&
                &.false.)
```

```
            ! convert to X, Y, Z
            xyz_plot(:,:,2) = xyz_plot(:,:,1)*sin(ang_plot(:,:,2))
            xyz_plot(:,:,1) = xyz_plot(:,:,1)*cos(ang_plot(:,:,2))

            ! print
            call print_ex_3d('plasma boundary',trim(plot_name),xyz_plot(:,:,3),&
                &x=xyz_plot(:,:,1),y=xyz_plot(:,:,2),draw=.false.)
            call draw_ex(['plasma boundary'],trim(plot_name),1,2,.false.)
        end subroutine plot_boundary

        ! print the mode numbers
        !> \private
        integer function print_mode_numbers(file_i,B,n_pert,max_n_B_output) &
            &result(ierr)
            character(*), parameter :: rout_name = 'print_mode_numbers'

            ! input / output
            integer, intent(in) :: file_i                                   ! file to output flux quantities for VMEC export
            real(dp), intent(in) :: b(:,:,:)                                ! cosine and sine of fourier series for R and Z
            integer, intent(in) :: n_pert                                   ! toroidal mode number
            integer, intent(in) :: max_n_b_output                           ! maximum number of modes to output

            ! local variables
            integer :: kd                                                   ! counter
            integer :: m                                                    ! counter
            character :: var_name                                           ! name of variables (R or Z)
            character(len=2*max_str_ln) :: temp_output_str                  ! temporary output string

            ! initialize ierr
            ierr = 0

            do kd = 1,2                                                     ! R and Z
                select case (kd)
                    case (1)
                        var_name = 'R'
                    case (2)
                        var_name = 'Z'
                    case default
                        var_name = '?'
                end select

                ! output to file
                do m = 0,min(size(b,1)-1,max_n_b_output)
                    write(temp_output_str,"(' ',A,'BC(',I4,', ',I4,') = ',&
                        &ES23.16,'   ',A,'BS(',I4,', ',I4,') = ',ES23.16)") &
                        &var_name,n_pert,m,b(m+1,1,kd), &
                        &var_name,n_pert,m,b(m+1,2,kd)
                    write(unit=file_i,fmt='(A)',iostat=ierr) &
                        &trim(temp_output_str)
                    chckerr('Failed to write')
                end do
                write(unit=file_i,fmt="(A)",iostat=ierr) ""
                chckerr('Failed to write')
            end do
        end function print_mode_numbers
    end function create_vmec_input

    !> \private flux version
    integer function print_output_eq_1(grid_eq,eq,data_name) result(ierr)
        use num_vars, only: pb3d_name
        use hdf5_ops, only: print_hdf5_arrs
        use hdf5_vars, only: dealloc_var_1d, var_1d_type, &
            &max_dim_var_1d
        use grid_utilities, only: trim_grid

        character(*), parameter :: rout_name = 'print_output_eq_1'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                              !< equilibrium grid variables
        type(eq_1_type), intent(in) :: eq                                   !< flux equilibrium variables
        character(len=*), intent(in) :: data_name                           !< name under which to store

        ! local variables
        integer :: norm_id(2)                                               ! untrimmed normal indices for trimmed grids
        type(var_1d_type), allocatable, target :: eq_1d(:)                  ! 1D equivalent of eq. variables
        type(var_1d_type), pointer :: eq_1d_loc => null()                   ! local element in eq_1D
        type(grid_type) :: grid_trim                                        ! trimmed grid
        integer :: id                                                       ! counter
        integer :: loc_size                                                 ! local size

        ! initialize ierr
        ierr = 0

        ! user output
```

```
        call writo('Write flux equilibrium variables to output file')
        call lvl_ud(1)

        ! trim grids
        ierr = trim_grid(grid_eq,grid_trim,norm_id)
        chckerr('')

        ! set up the 1D equivalents of the equilibrium variables
        allocate(eq_1d(max_dim_var_1d))

        ! Set up common variables eq_1D
        id = 1

        ! pres_FD
        eq_1d_loc => eq_1d(id); id = id+1
        eq_1d_loc%var_name = 'pres_FD'
        allocate(eq_1d_loc%tot_i_min(2),eq_1d_loc%tot_i_max(2))
        allocate(eq_1d_loc%loc_i_min(2),eq_1d_loc%loc_i_max(2))
        eq_1d_loc%tot_i_min = [1,0]
        eq_1d_loc%tot_i_max = [grid_trim%n(3),size(eq%pres_FD,2)-1]
        eq_1d_loc%loc_i_min = [grid_trim%i_min,0]
        eq_1d_loc%loc_i_max = [grid_trim%i_max,size(eq%pres_FD,2)-1]
        loc_size = size(eq%pres_FD(norm_id(1):norm_id(2),:))
        allocate(eq_1d_loc%p(loc_size))
        eq_1d_loc%p = reshape(eq%pres_FD(norm_id(1):norm_id(2),:),[loc_size])

        ! q_saf_FD
        eq_1d_loc => eq_1d(id); id = id+1
        eq_1d_loc%var_name = 'q_saf_FD'
        allocate(eq_1d_loc%tot_i_min(2),eq_1d_loc%tot_i_max(2))
        allocate(eq_1d_loc%loc_i_min(2),eq_1d_loc%loc_i_max(2))
        eq_1d_loc%tot_i_min = [1,0]
        eq_1d_loc%tot_i_max = [grid_trim%n(3),size(eq%q_saf_FD,2)-1]
        eq_1d_loc%loc_i_min = [grid_trim%i_min,0]
        eq_1d_loc%loc_i_max = [grid_trim%i_max,size(eq%q_saf_FD,2)-1]
        loc_size = size(eq%q_saf_FD(norm_id(1):norm_id(2),:))
        allocate(eq_1d_loc%p(loc_size))
        eq_1d_loc%p = reshape(eq%q_saf_FD(norm_id(1):norm_id(2),:),[loc_size])

        ! rot_t_FD
        eq_1d_loc => eq_1d(id); id = id+1
        eq_1d_loc%var_name = 'rot_t_FD'
        allocate(eq_1d_loc%tot_i_min(2),eq_1d_loc%tot_i_max(2))
        allocate(eq_1d_loc%loc_i_min(2),eq_1d_loc%loc_i_max(2))
        eq_1d_loc%tot_i_min = [1,0]
        eq_1d_loc%tot_i_max = [grid_trim%n(3),size(eq%rot_t_FD,2)-1]
        eq_1d_loc%loc_i_min = [grid_trim%i_min,0]
        eq_1d_loc%loc_i_max = [grid_trim%i_max,size(eq%rot_t_FD,2)-1]
        loc_size = size(eq%rot_t_FD(norm_id(1):norm_id(2),:))
        allocate(eq_1d_loc%p(loc_size))
        eq_1d_loc%p = reshape(eq%rot_t_FD(norm_id(1):norm_id(2),:),[loc_size])

        ! flux_p_FD
        eq_1d_loc => eq_1d(id); id = id+1
        eq_1d_loc%var_name = 'flux_p_FD'
        allocate(eq_1d_loc%tot_i_min(2),eq_1d_loc%tot_i_max(2))
        allocate(eq_1d_loc%loc_i_min(2),eq_1d_loc%loc_i_max(2))
        eq_1d_loc%tot_i_min = [1,0]
        eq_1d_loc%tot_i_max = [grid_trim%n(3),size(eq%flux_p_FD,2)-1]
        eq_1d_loc%loc_i_min = [grid_trim%i_min,0]
        eq_1d_loc%loc_i_max = [grid_trim%i_max,size(eq%flux_p_FD,2)-1]
        loc_size = size(eq%flux_p_FD(norm_id(1):norm_id(2),:))
        allocate(eq_1d_loc%p(loc_size))
        eq_1d_loc%p = reshape(eq%flux_p_FD(norm_id(1):norm_id(2),:),[loc_size])

        ! flux_t_FD
        eq_1d_loc => eq_1d(id); id = id+1
        eq_1d_loc%var_name = 'flux_t_FD'
        allocate(eq_1d_loc%tot_i_min(2),eq_1d_loc%tot_i_max(2))
        allocate(eq_1d_loc%loc_i_min(2),eq_1d_loc%loc_i_max(2))
        eq_1d_loc%tot_i_min = [1,0]
        eq_1d_loc%tot_i_max = [grid_trim%n(3),size(eq%flux_t_FD,2)-1]
        eq_1d_loc%loc_i_min = [grid_trim%i_min,0]
        eq_1d_loc%loc_i_max = [grid_trim%i_max,size(eq%flux_t_FD,2)-1]
        loc_size = size(eq%flux_t_FD(norm_id(1):norm_id(2),:))
        allocate(eq_1d_loc%p(loc_size))
        eq_1d_loc%p = reshape(eq%flux_t_FD(norm_id(1):norm_id(2),:),[loc_size])

        ! rho
        eq_1d_loc => eq_1d(id); id = id+1
        eq_1d_loc%var_name = 'rho'
        allocate(eq_1d_loc%tot_i_min(1),eq_1d_loc%tot_i_max(1))
        allocate(eq_1d_loc%loc_i_min(1),eq_1d_loc%loc_i_max(1))
        eq_1d_loc%tot_i_min = 1
        eq_1d_loc%tot_i_max = grid_trim%n(3)
        eq_1d_loc%loc_i_min = grid_trim%i_min
        eq_1d_loc%loc_i_max = grid_trim%i_max
```

```
        loc_size = size(eq%rho(norm_id(1):norm_id(2)))
        allocate(eq_1d_loc%p(loc_size))
        eq_1d_loc%p = eq%rho(norm_id(1):norm_id(2))


        ! write
        ierr = print_hdf5_arrs(eq_1d(1:id-1),pb3d_name,trim(data_name),&
            &ind_print=.not.grid_trim%divided)
        chckerr('')

        ! clean up
        call grid_trim%dealloc()
        call dealloc_var_1d(eq_1d)
        nullify(eq_1d_loc)

        ! user output
        call lvl_ud(-1)
    end function print_output_eq_1
    !> \private metric version
    integer function print_output_eq_2(grid_eq,eq,data_name,rich_lvl,par_div,&
        &dealloc_vars) result(ierr)
        use num_vars, only: pb3d_name, eq_jobs_lims, eq_job_nr
        use hdf5_ops, only: print_hdf5_arrs
        use hdf5_vars, only: dealloc_var_1d, var_1d_type, &
            &max_dim_var_1d
        use grid_utilities, only: trim_grid

        character(*), parameter :: rout_name = 'print_output_eq_2'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                          !< equilibrium grid variables
        type(eq_2_type), intent(inout) :: eq                            !< metric equilibrium variables
        character(len=*), intent(in) :: data_name                       !< name under which to store
        integer, intent(in), optional :: rich_lvl                       !< Richardson level to print
        logical, intent(in), optional :: par_div                        !< is a parallely divided grid
        logical, intent(in), optional :: dealloc_vars                   !< deallocate variables on the fly after
    writing

        ! local variables
        integer :: n_tot(3)                                             ! total n
        integer :: par_id(2)                                            ! local parallel interval
        integer :: norm_id(2)                                           ! untrimmed normal indices for trimmed grids
        type(var_1d_type), allocatable, target :: eq_1d(:)              ! 1D equivalent of eq. variables
        type(var_1d_type), pointer :: eq_1d_loc => null()               ! local element in eq_1D
        type(grid_type) :: grid_trim                                    ! trimmed grid
        integer :: id                                                   ! counter
        logical :: par_div_loc                                          ! local par_div
        integer :: loc_size                                             ! local size
        logical :: dealloc_vars_loc                                     ! local dealloc_vars

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Write metric equilibrium variables to output file')
        call lvl_ud(1)

        ! trim grids
        ierr = trim_grid(grid_eq,grid_trim,norm_id)
        chckerr('')

        ! set local par_div
        par_div_loc = .false.
        if (present(par_div)) par_div_loc = par_div

        ! set total n and parallel interval
        n_tot = grid_trim%n
        par_id = [1,n_tot(1)]
        if (grid_trim%n(1).gt.0 .and. par_div_loc) then                 ! total grid includes all equilibrium jobs
            n_tot(1) = maxval(eq_jobs_lims)-minval(eq_jobs_lims)+1
            par_id = eq_jobs_lims(:,eq_job_nr)
        end if

        ! set up local dealloc_vars
        dealloc_vars_loc = .false.
        if (present(dealloc_vars)) dealloc_vars_loc = dealloc_vars

        ! set up the 1D equivalents of the equilibrium variables
        allocate(eq_1d(max_dim_var_1d))

        ! Set up common variables eq_1D
        id = 1

        ! g_FD
        eq_1d_loc => eq_1d(id); id = id+1
        eq_1d_loc%var_name = 'g_FD'
        allocate(eq_1d_loc%tot_i_min(7),eq_1d_loc%tot_i_max(7))
```

```
        allocate(eq_1d_loc%loc_i_min(7),eq_1d_loc%loc_i_max(7))
        eq_1d_loc%tot_i_min = [1,1,1,1,0,0,0]
        eq_1d_loc%tot_i_max = [n_tot,6,size(eq%g_FD,5)-1,size(eq%g_FD,6)-1,&
            &size(eq%g_FD,7)-1]
        eq_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,1,0,0,0]
        eq_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,6,&
            &size(eq%g_FD,5)-1,size(eq%g_FD,6)-1,size(eq%g_FD,7)-1]
        loc_size = size(eq%g_FD(:,:,norm_id(1):norm_id(2),:,:,:,:))
        allocate(eq_1d_loc%p(loc_size))
        eq_1d_loc%p = reshape(eq%g_FD(:,:,norm_id(1):norm_id(2),:,:,:,:),&
            &[loc_size])
        if (dealloc_vars_loc) deallocate(eq%g_FD)

        ! h_FD
        eq_1d_loc => eq_1d(id); id = id+1
        eq_1d_loc%var_name = 'h_FD'
        allocate(eq_1d_loc%tot_i_min(7),eq_1d_loc%tot_i_max(7))
        allocate(eq_1d_loc%loc_i_min(7),eq_1d_loc%loc_i_max(7))
        eq_1d_loc%tot_i_min = [1,1,1,1,0,0,0]
        eq_1d_loc%tot_i_max = [n_tot,6,size(eq%h_FD,5)-1,size(eq%h_FD,6)-1,&
            &size(eq%h_FD,7)-1]
        eq_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,1,0,0,0]
        eq_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,6,&
            &size(eq%h_FD,5)-1,size(eq%h_FD,6)-1,size(eq%h_FD,7)-1]
        loc_size = size(eq%h_FD(:,:,norm_id(1):norm_id(2),:,:,:,:))
        allocate(eq_1d_loc%p(loc_size))
        eq_1d_loc%p = reshape(eq%h_FD(:,:,norm_id(1):norm_id(2),:,:,:,:),&
            &[loc_size])
        if (dealloc_vars_loc) deallocate(eq%h_FD)

        ! jac_FD
        eq_1d_loc => eq_1d(id); id = id+1
        eq_1d_loc%var_name = 'jac_FD'
        allocate(eq_1d_loc%tot_i_min(6),eq_1d_loc%tot_i_max(6))
        allocate(eq_1d_loc%loc_i_min(6),eq_1d_loc%loc_i_max(6))
        eq_1d_loc%tot_i_min = [1,1,1,0,0,0]
        eq_1d_loc%tot_i_max = [n_tot,size(eq%jac_FD,4)-1,size(eq%jac_FD,5)-1,&
            &size(eq%jac_FD,6)-1]
        eq_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,0,0,0]
        eq_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,&
            &size(eq%jac_FD,4)-1,size(eq%jac_FD,5)-1,size(eq%jac_FD,6)-1]
        loc_size = size(eq%jac_FD(:,:,norm_id(1):norm_id(2),:,:,:,:))
        allocate(eq_1d_loc%p(loc_size))
        eq_1d_loc%p = reshape(eq%jac_FD(:,:,norm_id(1):norm_id(2),:,:,:,:),&
            &[loc_size])
        if (dealloc_vars_loc) deallocate(eq%jac_FD)

        ! S
        eq_1d_loc => eq_1d(id); id = id+1
        eq_1d_loc%var_name = 'S'
        allocate(eq_1d_loc%tot_i_min(3),eq_1d_loc%tot_i_max(3))
        allocate(eq_1d_loc%loc_i_min(3),eq_1d_loc%loc_i_max(3))
        eq_1d_loc%tot_i_min = [1,1,1]
        eq_1d_loc%tot_i_max = n_tot
        eq_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min]
        eq_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max]
        loc_size = size(eq%S(:,:,norm_id(1):norm_id(2)))
        allocate(eq_1d_loc%p(loc_size))
        eq_1d_loc%p = reshape(eq%S(:,:,norm_id(1):norm_id(2)),&
            &[loc_size])
        if (dealloc_vars_loc) deallocate(eq%S)

        ! kappa_n
        eq_1d_loc => eq_1d(id); id = id+1
        eq_1d_loc%var_name = 'kappa_n'
        allocate(eq_1d_loc%tot_i_min(3),eq_1d_loc%tot_i_max(3))
        allocate(eq_1d_loc%loc_i_min(3),eq_1d_loc%loc_i_max(3))
        eq_1d_loc%tot_i_min = [1,1,1]
        eq_1d_loc%tot_i_max = n_tot
        eq_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min]
        eq_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max]
        loc_size = size(eq%kappa_n(:,:,norm_id(1):norm_id(2)))
        allocate(eq_1d_loc%p(loc_size))
        eq_1d_loc%p = reshape(eq%kappa_n(:,:,norm_id(1):norm_id(2)),&
            &[loc_size])
        if (dealloc_vars_loc) deallocate(eq%kappa_n)

        ! kappa_g
        eq_1d_loc => eq_1d(id); id = id+1
        eq_1d_loc%var_name = 'kappa_g'
        allocate(eq_1d_loc%tot_i_min(3),eq_1d_loc%tot_i_max(3))
        allocate(eq_1d_loc%loc_i_min(3),eq_1d_loc%loc_i_max(3))
        eq_1d_loc%tot_i_min = [1,1,1]
        eq_1d_loc%tot_i_max = n_tot
        eq_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min]
        eq_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max]
        loc_size = size(eq%kappa_g(:,:,norm_id(1):norm_id(2)))
```

```
        allocate(eq_1d_loc%p(loc_size))
        eq_1d_loc%p = reshape(eq%kappa_g(:,:,norm_id(1):norm_id(2)),&
            &[loc_size])
        if (dealloc_vars_loc) deallocate(eq%kappa_g)

        ! sigma
        eq_1d_loc => eq_1d(id); id = id+1
        eq_1d_loc%var_name = 'sigma'
        allocate(eq_1d_loc%tot_i_min(3),eq_1d_loc%tot_i_max(3))
        allocate(eq_1d_loc%loc_i_min(3),eq_1d_loc%loc_i_max(3))
        eq_1d_loc%tot_i_min = [1,1,1]
        eq_1d_loc%tot_i_max = n_tot
        eq_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min]
        eq_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max]
        loc_size = size(eq%sigma(:,:,norm_id(1):norm_id(2)))
        allocate(eq_1d_loc%p(loc_size))
        eq_1d_loc%p = reshape(eq%sigma(:,:,norm_id(1):norm_id(2)),&
            &[loc_size])
        if (dealloc_vars_loc) deallocate(eq%sigma)

        ! write
        ierr = print_hdf5_arrs(eq_1d(1:id-1),pb3d_name,trim(data_name),&
            &rich_lvl=rich_lvl,ind_print=.not.grid_trim%divided)
        chckerr('')

        ! clean up
        call grid_trim%dealloc()
        call dealloc_var_1d(eq_1d)
        nullify(eq_1d_loc)

        ! user output
        call lvl_ud(-1)
    end function print_output_eq_2

    !> \private flux version
    integer function redistribute_output_eq_1(grid,grid_out,eq,eq_out) &
        &result(ierr)
        use mpi_utilities, only: redistribute_var

        character(*), parameter :: rout_name = 'redistribute_output_eq_1'

        ! input / output
        type(grid_type), intent(in) :: grid                                     !< equilibrium grid variables
        type(grid_type), intent(in) :: grid_out                                 !< redistributed equilibrium grid variables
        type(eq_1_type), intent(in) :: eq                                       !< flux equilibrium variables
        type(eq_1_type), intent(inout) :: eq_out                                !< flux equilibrium variables in
redistributed grid

        ! local variables
        integer :: id                                                           ! counter

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Redistribute flux equilibrium variables')
        call lvl_ud(1)

        ! test
        if (grid%n(1).ne.grid_out%n(1) .or. grid%n(2).ne.grid_out%n(2)) then
            ierr = 1
            chckerr('grid and grid_out are not compatible')
        end if

        ! create redistributed flux equilibrium variables
        call eq_out%init(grid_out,setup_e=.false.,setup_f=.true.)

        ! for all derivatives
        do id = 0,max_deriv+1
            ! pres_FD
            ierr = redistribute_var(eq%pres_FD(:,id),eq_out%pres_FD(:,id),&
                &[grid%i_min,grid%i_max],[grid_out%i_min,grid_out%i_max])
            chckerr('')

            ! q_saf_FD
            ierr = redistribute_var(eq%q_saf_FD(:,id),eq_out%q_saf_FD(:,id),&
                &[grid%i_min,grid%i_max],[grid_out%i_min,grid_out%i_max])
            chckerr('')

            ! rot_t_FD
            ierr = redistribute_var(eq%rot_t_FD(:,id),eq_out%rot_t_FD(:,id),&
                &[grid%i_min,grid%i_max],[grid_out%i_min,grid_out%i_max])
            chckerr('')

            ! flux_p_FD
            ierr = redistribute_var(eq%flux_p_FD(:,id),eq_out%flux_p_FD(:,id),&
                &[grid%i_min,grid%i_max],[grid_out%i_min,grid_out%i_max])
```

```
            chckerr('')

            ! flux_t_FD
            ierr = redistribute_var(eq%flux_t_FD(:,id),eq_out%flux_t_FD(:,id),&
                &[grid%i_min,grid%i_max],[grid_out%i_min,grid_out%i_max])
            chckerr('')
        end do

        ! rho
        ierr = redistribute_var(eq%rho,eq_out%rho,&
            &[grid%i_min,grid%i_max],[grid_out%i_min,grid_out%i_max])
        chckerr('')

        ! user output
        call lvl_ud(-1)
    end function redistribute_output_eq_1
    !> \private metric version
    integer function redistribute_output_eq_2(grid,grid_out,eq,eq_out) &
        &result(ierr)
        use mpi_utilities, only: redistribute_var

        character(*), parameter :: rout_name = 'redistribute_output_eq_2'

        ! input / output
        type(grid_type), intent(in) :: grid                                     !< equilibrium grid variables
        type(grid_type), intent(in) :: grid_out                                 !< redistributed equilibrium grid variables
        type(eq_2_type), intent(in) :: eq                                       !< metric equilibrium variables
        type(eq_2_type), intent(inout) :: eq_out                                !< metric equilibrium variables in redistributed grid

        ! local variables
        integer :: id, jd, kd, ld                                               ! counters
        integer :: lims(2), lims_dis(2)                                         ! limits and distributed limits, taking into account the angular extent
        integer :: siz(3), siz_dis(3)                                           ! size for geometric part of variable
        real(dp), allocatable :: temp_var(:)                                    ! temporary variable

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Redistribute metric equilibrium variables')
        call lvl_ud(1)

        ! test
        if (grid%n(1).ne.grid_out%n(1) .or. grid%n(2).ne.grid_out%n(2)) then
            ierr = 1
            chckerr('grid and grid_out are not compatible')
        end if

        ! create redistributed metric equilibrium variables
        call eq_out%init(grid_out,setup_e=.false.,setup_f=.true.)

        ! set up limits taking into account angular extent and temporary var
        lims(1) = product(grid%n(1:2))*(grid%i_min-1)+1
        lims(2) = product(grid%n(1:2))*grid%i_max
        lims_dis(1) = product(grid%n(1:2))*(grid_out%i_min-1)+1
        lims_dis(2) = product(grid%n(1:2))*grid_out%i_max
        siz = [grid%n(1:2),grid%loc_n_r]
        siz_dis = [grid_out%n(1:2),grid_out%loc_n_r]
        allocate(temp_var(product(siz_dis)))

        ! for all derivatives and metric factors
        do jd = 0,max_deriv
            do kd = 0,max_deriv
                do ld = 0,max_deriv
                    do id = 1,size(eq%g_FD,4)
                        ! g_FD
                        ierr = redistribute_var(reshape(&
                            &eq%g_FD(:,:,:,id,jd,kd,ld),[product(siz)]),&
                            &temp_var,lims,lims_dis)
                        chckerr('')
                        eq_out%g_FD(:,:,:,id,jd,kd,ld) = &
                            &reshape(temp_var,siz_dis)

                        ! h_FD
                        ierr = redistribute_var(reshape(&
                            &eq%h_FD(:,:,:,id,jd,kd,ld),[product(siz)]),&
                            &temp_var,lims,lims_dis)
                        chckerr('')
                        eq_out%h_FD(:,:,:,id,jd,kd,ld) = &
                            &reshape(temp_var,siz_dis)
                    end do
                    ! jac_FD
                    ierr = redistribute_var(reshape(&
                        &eq%jac_FD(:,:,:,jd,kd,ld),[product(siz)]),&
                        &temp_var,lims,lims_dis)
```

```
                    chckerr('')
                    eq_out%jac_FD(:,:,:,jd,kd,ld) = &
                        &reshape(temp_var,siz_dis)
                end do
            end do
        end do

        ! S
        ierr = redistribute_var(reshape(eq%S,[product(siz)]),temp_var,&
            &lims,lims_dis)
        chckerr('')
        eq_out%S = reshape(temp_var,siz_dis)

        ! kappa_n
        ierr = redistribute_var(reshape(eq%kappa_n,[product(siz)]),temp_var,&
            &lims,lims_dis)
        chckerr('')
        eq_out%kappa_n = reshape(temp_var,siz_dis)

        ! kappa_g
        ierr = redistribute_var(reshape(eq%kappa_g,[product(siz)]),temp_var,&
            &lims,lims_dis)
        chckerr('')
        eq_out%kappa_g = reshape(temp_var,siz_dis)

        ! sigma
        ierr = redistribute_var(reshape(eq%sigma,[product(siz)]),temp_var,&
            &lims,lims_dis)
        chckerr('')
        eq_out%sigma = reshape(temp_var,siz_dis)

        ! user output
        call lvl_ud(-1)
    end function redistribute_output_eq_2

    !> \private individual version
    integer function calc_rzl_ind(grid_eq,eq,deriv) result(ierr)
        use vmec_utilities, only: fourier2real
        use vmec_vars, only: r_v_c, r_v_s, z_v_c, z_v_s, l_v_c, l_v_s, is_asym_v

        character(*), parameter :: rout_name = 'calc_RZL_ind'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(eq_2_type), intent(inout) :: eq                                    !< metric equilibrium
        integer, intent(in) :: deriv(3)                                         !< derivatives

        ! initialize ierr
        ierr = 0

#if ldebug
        ! check the derivatives requested
        ierr = check_deriv(deriv,max_deriv+1,'calc_RZL')
        chckerr('')
#endif

        ! calculate the variables R,Z and their derivatives
        ierr = fourier2real(r_v_c(:,grid_eq%i_min:grid_eq%i_max,deriv(1)),&
            &r_v_s(:,grid_eq%i_min:grid_eq%i_max,deriv(1)),&
            &grid_eq%trigon_factors,eq%R_E(:,:,:,deriv(1),deriv(2),deriv(3)),&
            &sym=[.true.,is_asym_v],deriv=[deriv(2),deriv(3)])
        chckerr('')
        ierr = fourier2real(z_v_c(:,grid_eq%i_min:grid_eq%i_max,deriv(1)),&
            &z_v_s(:,grid_eq%i_min:grid_eq%i_max,deriv(1)),&
            &grid_eq%trigon_factors,eq%Z_E(:,:,:,deriv(1),deriv(2),deriv(3)),&
            &sym=[is_asym_v,.true.],deriv=[deriv(2),deriv(3)])
        chckerr('')
        ierr = fourier2real(l_v_c(:,grid_eq%i_min:grid_eq%i_max,deriv(1)),&
            &l_v_s(:,grid_eq%i_min:grid_eq%i_max,deriv(1)),&
            &grid_eq%trigon_factors,eq%L_E(:,:,:,deriv(1),deriv(2),deriv(3)),&
            &sym=[is_asym_v,.true.],deriv=[deriv(2),deriv(3)])
        chckerr('')
    end function calc_rzl_ind
    !> \private array version
    integer function calc_rzl_arr(grid_eq,eq,deriv) result(ierr)
        character(*), parameter :: rout_name = 'calc_RZL_arr'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(eq_2_type), intent(inout) :: eq                                    !< metric equilibrium
        integer, intent(in) :: deriv(:,:)                                       !< derivatives

        ! local variables
        integer :: id

        ! initialize ierr
        ierr = 0
```

```

        do id = 1, size(deriv,2)
            ierr = calc_rzl_ind(grid_eq,eq,deriv(:,id))
            chckerr('')
        end do
    end function calc_rzl_arr

    !> \private individual version
    integer function calc_g_c_ind(eq,deriv) result(ierr)
        use num_utilities, only: add_arr_mult, c

        character(*), parameter :: rout_name = 'calc_g_C_ind'

        ! input / output
        type(eq_2_type), intent(inout) :: eq                                    !< metric equilibrium
        integer, intent(in) :: deriv(:)                                         !< derivatives

        ! initialize ierr
        ierr = 0

#if ldebug
        ! check the derivatives requested
        ierr = check_deriv(deriv,max_deriv,'calc_g_C')
        chckerr('')
#endif

        ! initialize
        eq%g_C(:,:,:,:,deriv(1),deriv(2),deriv(3)) = 0._dp

        ! calculate
        if (sum(deriv).eq.0) then
            eq%g_C(:,:,:,c([1,1],.true.),deriv(1),deriv(2),deriv(3)) = 1.0_dp
            eq%g_C(:,:,:,c([3,3],.true.),deriv(1),deriv(2),deriv(3)) = 1.0_dp
        end if
        ierr = add_arr_mult(eq%R_E,eq%R_E,&
            &eq%g_C(:,:,:,c([2,2],.true.),deriv(1),deriv(2),deriv(3)),deriv)
        chckerr('')
    end function calc_g_c_ind
    !> \private array version
    integer function calc_g_c_arr(eq,deriv) result(ierr)
        character(*), parameter :: rout_name = 'calc_g_C_arr'

        ! input / output
        type(eq_2_type), intent(inout) :: eq                                    !< metric equilibrium
        integer, intent(in) :: deriv(:,:)                                       !< derivatives

        ! local variables
        integer :: id

        ! initialize ierr
        ierr = 0

        do id = 1, size(deriv,2)
            ierr = calc_g_c_ind(eq,deriv(:,id))
            chckerr('')
        end do
    end function calc_g_c_arr

    !> \private individual version
    integer function calc_g_v_ind(eq,deriv) result(ierr)
        use eq_utilities, only: calc_g

        character(*), parameter :: rout_name = 'calc_g_V_ind'

        ! input / output
        type(eq_2_type), intent(inout) :: eq                                    !< metric equilibrium
        integer, intent(in) :: deriv(:)                                         !< derivatives

        ! initialize ierr
        ierr = 0

#if ldebug
        ! check the derivatives requested
        ierr = check_deriv(deriv,max_deriv,'calc_g_V')
        chckerr('')
#endif

        ierr = calc_g(eq%g_C,eq%T_VC,eq%g_E,deriv,max_deriv)
        chckerr('')
    end function calc_g_v_ind
    !> \private  array version
    integer function calc_g_v_arr(eq,deriv) result(ierr)
        character(*), parameter :: rout_name = 'calc_g_V_arr'

        ! input / output
        type(eq_2_type), intent(inout) :: eq                                    !< metric equilibrium
        integer, intent(in) :: deriv(:,:)                                       !< derivatives
```

```fortran

        ! local variables
        integer :: id

        ! initialize ierr
        ierr = 0

        do id = 1, size(deriv,2)
            ierr = calc_g_v_ind(eq,deriv(:,id))
            chckerr('')
        end do
    end function calc_g_v_arr

    !> \private individual version
    integer function calc_g_h_ind(grid_eq,eq,deriv) result(ierr)
        use num_vars, only: norm_disc_prec_eq
        use helena_vars, only: ias, nchi, r_h, z_h, chi_h, h_h_33
        use num_utilities, only: c, spline
        use ezspline_obj
        use ezspline

        character(*), parameter :: rout_name = 'calc_g_H_ind'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(eq_2_type), intent(inout) :: eq                                    !< metric equilibrium
        integer, intent(in) :: deriv(:)                                         !< derivatives

        ! local variables
        type(ezspline2_r8) :: f_spl(2)                                          ! spline object for interpolation, even and odd
        character(len=max_str_ln) :: err_msg                                    ! error message
        integer :: id, jd, kd, ld                                               ! counters
        integer :: bcs(2,3)                                                     ! boundary conditions (theta(even), theta(odd), r)
        integer :: bc_ld(6)                                                     ! boundary condition type for each metric element
        real(dp) :: bcs_val(2,3)                                                ! values for boundary conditions
        real(dp), allocatable :: rchi(:,:), rpsi(:,:)                           ! chi and psi derivatives of R
        real(dp), allocatable :: zchi(:,:), zpsi(:,:)                           ! chi and psi derivatives of Z

        ! initialize ierr
        ierr = 0

#if ldebug
        ! check the derivatives requested
        ierr = check_deriv(deriv,max_deriv,'calc_g_H')
        chckerr('')
#endif

        ! set up boundary conditions
        if (ias.eq.0) then                                                      ! top-bottom symmetric
            bcs(:,1) = [1,1]                                                    ! theta(even): zero first derivative
            bcs(:,2) = [2,2]                                                    ! theta(odd): zero first derivative
        else
            bcs(:,1) = [-1,-1]                                                  ! theta(even): periodic
            bcs(:,2) = [-1,-1]                                                  ! theta(odd): periodic
        end if
        bcs(:,3) = [0,0]                                                        ! r: not-a-knot
        bcs_val = 0._dp

        ! boundary condition type for each metric element
        bc_ld = [1,2,0,1,0,1]                                                   ! 1: even, 2: odd, 0: zero

        ! initialize
        eq%g_E(:,:,:,:,deriv(1),deriv(2),deriv(3)) = 0._dp

        if (sum(deriv).eq.0) then                                               ! no derivatives
            ! initialize variables
            allocate(rchi(nchi,grid_eq%loc_n_r),rpsi(nchi,grid_eq%loc_n_r))
            allocate(zchi(nchi,grid_eq%loc_n_r),zpsi(nchi,grid_eq%loc_n_r))

            ! normal derivatives
            do id = 1,grid_eq%n(1)
                ierr = spline(grid_eq%loc_r_E,&
                    &r_h(id,grid_eq%i_min:grid_eq%i_max),grid_eq%loc_r_E,&
                    &rpsi(id,:),ord=norm_disc_prec_eq,deriv=1,bcs=bcs(:,3),&
                    &bcs_val=bcs_val(:,3))
                chckerr('')
                ierr = spline(grid_eq%loc_r_E,&
                    &z_h(id,grid_eq%i_min:grid_eq%i_max),grid_eq%loc_r_E,&
                    &zpsi(id,:),ord=norm_disc_prec_eq,deriv=1,bcs=bcs(:,3),&
                    &bcs_val=bcs_val(:,3))
                chckerr('')
            end do

            ! poloidal derivatives
```

```
            do kd = 1,grid_eq%loc_n_r
                ierr = spline(chi_h,r_h(:,grid_eq%i_min-1+kd),chi_h,&
                    &rchi(:,kd),ord=norm_disc_prec_eq,deriv=1,bcs=bcs(:,1),&
                    &bcs_val=bcs_val(:,1))                                  ! even
                chckerr('')
                ierr = spline(chi_h,z_h(:,grid_eq%i_min-1+kd),chi_h,&
                    &zchi(:,kd),ord=norm_disc_prec_eq,deriv=1,bcs=bcs(:,2),&
                    &bcs_val=bcs_val(:,2))                                  ! odd
                chckerr('')
            end do

            ! set up g_H
            do jd = 1,grid_eq%n(2)
                eq%g_E(:,jd,:,c([1,1],.true.),0,0,0) = rpsi*rpsi+zpsi*zpsi
                eq%g_E(:,jd,:,c([1,2],.true.),0,0,0) = rpsi*rchi+zpsi*zchi
                eq%g_E(:,jd,:,c([2,2],.true.),0,0,0) = rpsi*rchi+zpsi*zchi
                eq%g_E(:,jd,:,c([2,2],.true.),0,0,0) = rchi*rchi+zchi*zchi
                eq%g_E(:,jd,:,c([3,3],.true.),0,0,0) = &
                    &1._dp/h_h_33(:,grid_eq%i_min:grid_eq%i_max)
            end do

            ! clean up
            deallocate(zchi,rchi,zpsi,rpsi)
        else if (deriv(3).ne.0) then                                        ! axisymmetry: deriv. in tor. coord. is zero
            !eq%g_E(:,:,:,:,deriv(1),deriv(2),deriv(3)) = 0.0_dp
        else if (deriv(1).le.2 .and. deriv(2).le.2) then
            ! initialize 2-D cubic spline for even (1) and odd (2) quantities
            do ld = 1,2
                call ezspline_init(f_spl(ld),grid_eq%n(1),grid_eq%loc_n_r,&
                    &bcs(:,ld),bcs(:,3),ierr)
                call ezspline_error(ierr)
                chckerr('')

                ! set grid
                f_spl(ld)%x1 = chi_h
                f_spl(ld)%x2 = grid_eq%loc_r_E

                ! set boundary conditions
                f_spl(ld)%bcval1min = bcs_val(1,ld)
                f_spl(ld)%bcval1max = bcs_val(2,ld)
                f_spl(ld)%bcval2min = bcs_val(1,3)
                f_spl(ld)%bcval2max = bcs_val(2,3)
            end do

            do ld = 1,6
                do jd = 1,grid_eq%n(2)
                    if (bc_ld(ld).eq.0) cycle                               ! quantity is zero

                    ! set up
                    call ezspline_setup(f_spl(bc_ld(ld)),&
                        &eq%g_E(:,jd,:,ld,0,0,0),ierr,exact_dim=.true.)     ! match exact dimensions, none of them old
Fortran bullsh*t!
                    call ezspline_error(ierr)
                    chckerr('')

                    ! derivate
                    ! Note:  deriv  is the  other  way  around because  poloidal
                    ! indices come before normal
                    call ezspline_derivative(f_spl(bc_ld(ld)),deriv(2),&
                        &deriv(1),grid_eq%n(1),grid_eq%loc_n_r,chi_h,&
                        &grid_eq%loc_r_E,eq%g_E(:,jd,:,ld,deriv(1),deriv(2),0),&
                        &ierr)
                    call ezspline_error(ierr)
                    chckerr('')
                end do
            end do

            ! free
            do ld = 1,2
                call ezspline_free(f_spl(ld),ierr)
                call ezspline_error(ierr)
            end do
        else
            ierr = 1
            err_msg = 'Derivative of order ('//trim(i2str(deriv(1)))//','//&
                &trim(i2str(deriv(2)))//','//trim(i2str(deriv(3)))//'&
                &) not supported'
            chckerr(err_msg)
        end if
    end function calc_g_h_ind
    !> \private array version
    integer function calc_g_h_arr(grid_eq,eq,deriv) result(ierr)
        character(*), parameter :: rout_name = 'calc_g_H_arr'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                              !< equilibirum grid
        type(eq_2_type), intent(inout) :: eq                                !< metric equilibrium
```

```
        integer, intent(in) :: deriv(:,:)                                   !< derivatives

        ! local variables
        integer :: id

        ! initialize ierr
        ierr = 0

        do id = 1, size(deriv,2)
            ierr = calc_g_h_ind(grid_eq,eq,deriv(:,id))
            chckerr('')
        end do
    end function calc_g_h_arr

    !> \private individual version
    integer function calc_g_f_ind(eq,deriv) result(ierr)
        use eq_utilities, only: calc_g

        character(*), parameter :: rout_name = 'calc_g_F_ind'

        ! input / output
        type(eq_2_type), intent(inout) :: eq                                !< metric equilibrium
        integer, intent(in) :: deriv(:)                                     !< derivatives

        ! initialize ierr
        ierr = 0

#if ldebug
        ! check the derivatives requested
        ierr = check_deriv(deriv,max_deriv,'calc_g_F')
        chckerr('')
#endif

        ! calculate g_F
        ierr = calc_g(eq%g_E,eq%T_FE,eq%g_F,deriv,max_deriv)
        chckerr('')
    end function calc_g_f_ind
    !> \private array version
    integer function calc_g_f_arr(eq,deriv) result(ierr)
        character(*), parameter :: rout_name = 'calc_g_F_arr'

        ! input / output
        type(eq_2_type), intent(inout) :: eq                                !< metric equilibrium
        integer, intent(in) :: deriv(:,:)                                   !< derivatives

        ! local variables
        integer :: id

        ! initialize ierr
        ierr = 0

        do id = 1, size(deriv,2)
            ierr = calc_g_f_ind(eq,deriv(:,id))
            chckerr('')
        end do
    end function calc_g_f_arr

    !> \private individual version
    integer function calc_jac_v_ind(grid,eq,deriv) result(ierr)
        use vmec_utilities, only: fourier2real
        use vmec_vars, only: jac_v_c, jac_v_s, is_asym_v

        character(*), parameter :: rout_name = 'calc_jac_V_ind'

        ! input / output
        type(grid_type), intent(in) :: grid                                 !< grid for which to calculate Jacobian
        type(eq_2_type), intent(inout) :: eq                                !< metric equilibrium
        integer, intent(in) :: deriv(:)                                     !< derivatives

        ! initialize ierr
        ierr = 0

#if ldebug
        ! check the derivatives requested
        ierr = check_deriv(deriv,max_deriv,'calc_J_V')
        chckerr('')
#endif

        ! initialize
        eq%jac_E(:,:,:,deriv(1),deriv(2),deriv(3)) = 0.0_dp

        ! calculate the Jacobian and its derivatives
        ierr = fourier2real(jac_v_c(:,grid%i_min:grid%i_max,deriv(1)),&
            &jac_v_s(:,grid%i_min:grid%i_max,deriv(1)),grid%trigon_factors,&
            &eq%jac_E(:,:,:,deriv(1),deriv(2),deriv(3)),&
            &sym=[.true.,is_asym_v],deriv=[deriv(2),deriv(3)])
        chckerr('')
```

```
    end function calc_jac_v_ind
    !> \private array version
    integer function calc_jac_v_arr(grid,eq,deriv) result(ierr)
        character(*), parameter :: rout_name = 'calc_jac_V_arr'

        ! input / output
        type(grid_type), intent(in) :: grid                                     !< grid for which to calculate Jacobian
        type(eq_2_type), intent(inout) :: eq                                    !< metric equilibrium
        integer, intent(in) :: deriv(:,:)                                       !< derivatives

        ! local variables
        integer :: id

        ! initialize ierr
        ierr = 0

        do id = 1, size(deriv,2)
            ierr = calc_jac_v_ind(grid,eq,deriv(:,id))
            chckerr('')
        end do
    end function calc_jac_v_arr

    !> \private individual version
    integer function calc_jac_h_ind(grid_eq,eq_1,eq_2,deriv) result(ierr)
        use helena_vars, only: ias, h_h_33, rbphi_h, chi_h
        use ezspline_obj
        use ezspline

        character(*), parameter :: rout_name = 'calc_jac_H_ind'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(eq_1_type), intent(in) :: eq_1                                     !< flux equilibrium
        type(eq_2_type), intent(inout) :: eq_2                                  !< metric equilibrium
        integer, intent(in) :: deriv(:)                                         !< derivatives

        ! local variables
        character(len=max_str_ln) :: err_msg                                    ! error message
        type(ezspline2_r8) :: f_spl                                             ! spline object for interpolation
        integer :: jd, kd                                                       ! counters
        integer :: bcs(2,2)                                                     ! boundary conditions (theta(even), r)
        real(dp) :: bcs_val(2,2)                                                ! values for boundary conditions

        ! initialize ierr
        ierr = 0

#if ldebug
        ! check the derivatives requested
        ierr = check_deriv(deriv,max_deriv,'calc_jac_H')
        chckerr('')
#endif

        ! set up boundary conditions
        if (ias.eq.0) then                                                      ! top-bottom symmetric
            bcs(:,1) = [1,1]                                                    ! theta(even): zero first derivative
        else
            bcs(:,1) = [-1,-1]                                                  ! periodic
        end if
        bcs(:,2) = [0,0]                                                        ! r: not-a-knot
        bcs_val = 0._dp

        ! calculate determinant
        if (sum(deriv).eq.0) then                                               ! no derivatives
            do kd = 1,grid_eq%loc_n_r
                do jd = 1,grid_eq%n(2)
                    eq_2%jac_E(:,jd,kd,0,0,0) = eq_1%q_saf_E(kd,0)/&
                        &(h_h_33(:,grid_eq%i_min-1+kd)*&
                        &rbphi_h(grid_eq%i_min-1+kd,0))
                end do
            end do
        else if (deriv(3).ne.0) then                                            ! axisymmetry: deriv. in tor. coord. is zero
            eq_2%jac_E(:,:,:,deriv(1),deriv(2),deriv(3)) = 0.0_dp
        else if (deriv(1).le.2 .and. deriv(2).le.2) then
            ! initialize 2-D cubic spline
            call ezspline_init(f_spl,grid_eq%n(1),grid_eq%loc_n_r,&
                &bcs(:,1),bcs(:,2),ierr)
            call ezspline_error(ierr)
            chckerr('')


            ! set grid
            f_spl%x1 = chi_h
            f_spl%x2 = grid_eq%loc_r_E

            ! set boundary conditions
            f_spl%bcval1min = bcs_val(1,1)
            f_spl%bcval1max = bcs_val(2,1)
```

```
            f_spl%bcval2min = bcs_val(1,2)
            f_spl%bcval2max = bcs_val(2,2)

            do jd = 1,grid_eq%n(2)
                ! set up
                call ezspline_setup(f_spl,eq_2%jac_E(:,jd,:,0,0,0),ierr,&
                    &exact_dim=.true.)                                      ! match exact dimensions, none of them old Fortran bullsh*t!
                call ezspline_error(ierr)
                chckerr('')

                ! derivate
                ! Note: deriv is  the other way around  because poloidal indices
                ! come before normal
                call ezspline_derivative(f_spl,deriv(2),deriv(1),grid_eq%n(1),&
                    &grid_eq%loc_n_r,chi_h,grid_eq%loc_r_E,&
                    &eq_2%jac_E(:,jd,:,deriv(1),deriv(2),0),ierr)
                call ezspline_error(ierr)
                chckerr('')
            end do

            ! free
            call ezspline_free(f_spl,ierr)
            call ezspline_error(ierr)
        else
            ierr = 1
            err_msg = 'Derivative of order ('//trim(i2str(deriv(1)))//','//&
                &trim(i2str(deriv(2)))//','//trim(i2str(deriv(3)))//'&
                &) not supported'
            chckerr(err_msg)
        end if
    end function calc_jac_h_ind
    !> \private array version
    integer function calc_jac_h_arr(grid_eq,eq_1,eq_2,deriv) result(ierr)
        character(*), parameter :: rout_name = 'calc_jac_H_arr'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                              !< equilibrim grid
        type(eq_1_type), intent(in) :: eq_1                                 !< flux equilibrium
        type(eq_2_type), intent(inout) :: eq_2                              !< metric equilibrium
        integer, intent(in) :: deriv(:,:)                                   !< derivatives

        ! local variables
        integer :: id

        ! initialize ierr
        ierr = 0

        do id = 1, size(deriv,2)
            ierr = calc_jac_h_ind(grid_eq,eq_1,eq_2,deriv(:,id))
            chckerr('')
        end do
    end function calc_jac_h_arr

    !> \private individual version
    integer function calc_jac_f_ind(eq,deriv) result(ierr)
        use num_utilities, only: add_arr_mult

        character(*), parameter :: rout_name = 'calc_jac_F_ind'

        ! input / output
        type(eq_2_type), intent(inout) :: eq                                !< metric equilibrium
        integer, intent(in) :: deriv(:)                                     !< derivatives

        ! initialize ierr
        ierr = 0

#if ldebug
        ! check the derivatives requested
        ierr = check_deriv(deriv,max_deriv,'calc_J_F')
        chckerr('')
#endif

        ! initialize
        eq%jac_F(:,:,:,deriv(1),deriv(2),deriv(3)) = 0.0_dp

        ! calculate determinant
        ierr = add_arr_mult(eq%jac_E,eq%det_T_FE,&
            &eq%jac_F(:,:,:,deriv(1),deriv(2),deriv(3)),deriv)
        chckerr('')
    end function calc_jac_f_ind
    !> \private array version
    integer function calc_jac_f_arr(eq,deriv) result(ierr)
        character(*), parameter :: rout_name = 'calc_jac_F_arr'

        ! input / output
        type(eq_2_type), intent(inout) :: eq                                !< metric equilibrium
```

```
        integer, intent(in) :: deriv(:,:)                                   !< derivatives

        ! local variables
        integer :: id

        ! initialize ierr
        ierr = 0

        do id = 1, size(deriv,2)
            ierr = calc_jac_f_ind(eq,deriv(:,id))
            chckerr('')
        end do
    end function calc_jac_f_arr

    !> \private individual version
    integer function calc_t_vc_ind(eq,deriv) result(ierr)
        use num_utilities, only: add_arr_mult, c

        character(*), parameter :: rout_name = 'calc_T_VC_ind'

        ! input / output
        type(eq_2_type), intent(inout) :: eq                                !< metric equilibrium
        integer, intent(in) :: deriv(:)                                     !< derivatives


        ! initialize ierr
        ierr = 0
#if ldebug
        ! check the derivatives requested
        ierr = check_deriv(deriv,max_deriv,'calc_T_VC')
        chckerr('')
#endif

        ! initialize
        eq%T_VC(:,:,:,:,deriv(1),deriv(2),deriv(3)) = 0._dp

        ! calculate transformation matrix T_V^C
        eq%T_VC(:,:,:,c([1,1],.false.),deriv(1),deriv(2),deriv(3)) = &
            &eq%R_E(:,:,:,deriv(1)+1,deriv(2),deriv(3))
        !eq%T_VC(:,:,:,c([1,2],.false.),deriv(1),deriv(2),deriv(3)) = 0
        eq%T_VC(:,:,:,c([1,3],.false.),deriv(1),deriv(2),deriv(3)) = &
            &eq%Z_E(:,:,:,deriv(1)+1,deriv(2),deriv(3))
        eq%T_VC(:,:,:,c([2,1],.false.),deriv(1),deriv(2),deriv(3)) = &
            &eq%R_E(:,:,:,deriv(1),deriv(2)+1,deriv(3))
        !eq%T_VC(:,:,:,c([2,2],.false.),deriv(1),deriv(2),deriv(3)) = 0
        eq%T_VC(:,:,:,c([2,3],.false.),deriv(1),deriv(2),deriv(3)) = &
            &eq%Z_E(:,:,:,deriv(1),deriv(2)+1,deriv(3))
        eq%T_VC(:,:,:,c([3,1],.false.),deriv(1),deriv(2),deriv(3)) = &
            &eq%R_E(:,:,:,deriv(1),deriv(2),deriv(3)+1)
        if (sum(deriv).eq.0) then
            eq%T_VC(:,:,:,c([3,2],.false.),deriv(1),deriv(2),deriv(3)) = 1.0_dp
        else
            !eq%T_VC(:,:,:,c([3,2],.false.),deriv(1),deriv(2),deriv(3)) = 0
        end if
        eq%T_VC(:,:,:,c([3,3],.false.),deriv(1),deriv(2),deriv(3)) = &
            &eq%Z_E(:,:,:,deriv(1),deriv(2),deriv(3)+1)
    end function calc_t_vc_ind
    !> \private array version
    integer function calc_t_vc_arr(eq,deriv) result(ierr)
        character(*), parameter :: rout_name = 'calc_T_VC_arr'

        ! input / output
        type(eq_2_type), intent(inout) :: eq                                !< metric equilibrium
        integer, intent(in) :: deriv(:,:)                                   !< derivatives

        ! local variables
        integer :: id

        ! initialize ierr
        ierr = 0

        do id = 1, size(deriv,2)
            ierr = calc_t_vc_ind(eq,deriv(:,id))
            chckerr('')
        end do
    end function calc_t_vc_arr

    !> \private individual version
    integer function calc_t_vf_ind(grid_eq,eq_1,eq_2,deriv) result(ierr)
        use num_vars, only: use_pol_flux_f
        use num_utilities, only: add_arr_mult, c

        character(*), parameter :: rout_name = 'calc_T_VF_ind'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                              !< equilibrium grid
        type(eq_1_type), intent(in) :: eq_1                                 !< flux equilibrium
```

```
        type(eq_2_type), intent(inout) :: eq_2                                  !< metric equilibrium
        integer, intent(in) :: deriv(:)                                         !< derivatives

        ! local variables
        integer :: kd                                                           ! counter
        real(dp), allocatable :: theta_s(:,:,:,:,:,:)                           ! theta_F and derivatives
        real(dp), allocatable :: zeta_s(:,:,:,:,:,:)                            ! - zeta_F and derivatives
        integer :: dims(3)                                                      ! dimensions
        integer :: c1                                                           ! 2D coordinate in met_type storage convention

        ! initialize ierr
        ierr = 0

#if ldebug
        ! check the derivatives requested
        ierr = check_deriv(deriv,max_deriv,'calc_T_VF')
        chckerr('')
#endif

        ! set up dims
        dims = [grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r]

        ! initialize
        eq_2%T_EF(:,:,:,:,deriv(1),deriv(2),deriv(3)) = 0._dp

        ! set up theta_s
        allocate(theta_s(dims(1),dims(2),dims(3),&
            &0:deriv(1)+1,0:deriv(2)+1,0:deriv(3)+1))
        theta_s = 0.0_dp
        ! start from theta_E
        theta_s(:,:,:,0,0,0) = grid_eq%theta_E
        theta_s(:,:,:,0,1,0) = 1.0_dp
        ! add the deformation described by lambda
        theta_s = theta_s + &
            &eq_2%L_E(:,:,:,0:deriv(1)+1,0:deriv(2)+1,0:deriv(3)+1)

        if (use_pol_flux_f) then
            ! calculate transformation matrix T_V^F
            ! (1,1)
            ierr = add_arr_mult(theta_s,eq_1%q_saf_E(:,1:),&
                &eq_2%T_EF(:,:,:,c([1,1],.false.),deriv(1),deriv(2),deriv(3)),&
                &deriv)
            chckerr('')
            ierr = add_arr_mult(eq_2%L_E(:,:,:,1:,0:,0:),eq_1%q_saf_E(:,0:),&
                &eq_2%T_EF(:,:,:,c([1,1],.false.),deriv(1),deriv(2),deriv(3)),&
                &deriv)
            chckerr('')
            ! (1,2)
            if (deriv(2).eq.0 .and. deriv(3).eq.0) then
                do kd = 1,dims(3)
                    eq_2%T_EF(:,:,kd,c([1,2],.false.),deriv(1),0,0) = &
                        &eq_1%flux_p_E(kd,deriv(1)+1)/(2*pi)
                end do
            !else
                !eq_2%T_EF(:,:,:,c([1,2],.false.),deriv(1),deriv(2),deriv(3)) &
                    !&= 0.0_dp
            end if
            ! (1,3)
            eq_2%T_EF(:,:,:,c([1,3],.false.),deriv(1),deriv(2),deriv(3)) = &
                &eq_2%L_E(:,:,:,deriv(1)+1,deriv(2),deriv(3))
            ! (2,1)
            ierr = add_arr_mult(theta_s(:,:,:,0:,1:,0:),eq_1%q_saf_E,&
                &eq_2%T_EF(:,:,:,c([2,1],.false.),deriv(1),deriv(2),deriv(3)),&
                &deriv)
            chckerr('')
            ! (2,2)
            !eq_2%T_EF(:,:,:,c([2,2],.false.),deriv(1),deriv(2),deriv(3)) = &
                !&0.0_dp
            ! (2,3)
            eq_2%T_EF(:,:,:,c([2,3],.false.),deriv(1),deriv(2),deriv(3)) = &
                &theta_s(:,:,:,deriv(1),deriv(2)+1,deriv(3))
            ! (3,1)
            if (sum(deriv).eq.0) then
                eq_2%T_EF(:,:,:,c([3,1],.false.),0,0,0) = -1.0_dp
            end if
            ierr = add_arr_mult(eq_2%L_E(:,:,:,0:,0:,1:),eq_1%q_saf_E,&
                &eq_2%T_EF(:,:,:,c([3,1],.false.),deriv(1),deriv(2),deriv(3)),&
                &deriv)
            chckerr('')
            ! (3,2)
            !eq_2%T_EF(:,:,:,c([3,2],.false.),deriv(1),deriv(2),deriv(3)) = &
                !&0.0_dp
            ! (3,3)
            eq_2%T_EF(:,:,:,c([3,3],.false.),deriv(1),deriv(2),deriv(3)) = &
                &eq_2%L_E(:,:,:,deriv(1),deriv(2),deriv(3)+1)

```

```
            ! determinant
            eq_2%det_T_EF(:,:,:,deriv(1),deriv(2),deriv(3)) = 0.0_dp
            ierr = add_arr_mult(-theta_s(:,:,:,0:,1:,0:),&
                &eq_1%flux_p_E(:,1:)/(2*pi),&
                &eq_2%det_T_EF(:,:,:,deriv(1),deriv(2),deriv(3)),deriv)
            chckerr('')
        else
            ! set up zeta_s
            allocate(zeta_s(dims(1),dims(2),dims(3),0:deriv(1)+1,&
                &0:deriv(2)+1,0:deriv(3)+1))
            zeta_s = 0.0_dp
            ! start from zeta_E
            zeta_s(:,:,:,0,0,0) = grid_eq%zeta_E
            zeta_s(:,:,:,0,0,1) = 1.0_dp

            ! calculate transformation matrix T_V^F
            ! (1,1)
            eq_2%T_EF(:,:,:,c([1,1],.false.),deriv(1),deriv(2),deriv(3)) = &
                &-theta_s(:,:,:,deriv(1)+1,deriv(2),deriv(3))
            ierr = add_arr_mult(zeta_s,eq_1%rot_t_E(:,1:),&
                &eq_2%T_EF(:,:,:,c([1,1],.false.),deriv(1),deriv(2),deriv(3)),&
                &deriv)
            chckerr('')
            ! (1,2)
            if (deriv(2).eq.0 .and. deriv(3).eq.0) then
                do kd = 1,dims(3)
                    eq_2%T_EF(:,:,kd,c([1,2],.false.),deriv(1),0,0) = &
                        &-eq_1%flux_t_E(kd,deriv(1)+1)/(2*pi)
                end do
            !else
                !eq_2%T_EF(:,:,:,c([1,2],.false.),deriv(1),deriv(2),deriv(3)) 7
                    !&= 0.0_dp
            end if
            ! (1,3)
            !eq_2%T_EF(:,:,:,c([1,3],.false.),deriv(1),deriv(2),deriv(3)) = &
                !&0.0_dp
            ! (2,1)
            eq_2%T_EF(:,:,:,c([2,1],.false.),deriv(1),deriv(2),deriv(3)) = &
                &-theta_s(:,:,:,deriv(1),deriv(2)+1,deriv(3))
            ! (2,2)
            !eq_2%T_EF(:,:,:,c([2,2],.false.),deriv(1),deriv(2),deriv(3)) = &
                !&0.0_dp
            ! (2,3)
            !eq_2%T_EF(:,:,:,c([2,3],.false.),deriv(1),deriv(2),deriv(3)) = &
                !&0.0_dp
            ! (3,1)
            if (deriv(2).eq.0 .and. deriv(3).eq.0) then
                do kd = 1,dims(3)
                    eq_2%T_EF(:,:,kd,c([3,1],.false.),deriv(1),0,0) = &
                        &eq_1%rot_t_E(kd,deriv(1))
                end do
            end if
            c1 = c([3,1],.false.)                                          ! to avoid compiler crash (as of 26-02-2015)
            eq_2%T_EF(:,:,:,c1,deriv(1),deriv(2),deriv(3)) = &
                &eq_2%T_EF(:,:,:,c1,deriv(1),deriv(2),deriv(3)) &
                &-theta_s(:,:,:,deriv(1),deriv(2),deriv(3)+1)
            ! (3,2)
            !eq_2%T_EF(:,:,:,c([3,2],.false.),deriv(1),deriv(2),deriv(3)) = &
                !&0.0_dp
            ! (3,3)
            if (sum(deriv).eq.0) then
                eq_2%T_EF(:,:,:,c([3,3],.false.),0,0,0) = -1.0_dp
            !else
                !eq_2%T_EF(:,:,:,c([3,3],.false.)deriv(1),deriv(2),deriv(3)) = &
                    !&0.0_dp
            end if

            ! determinant
            eq_2%det_T_EF(:,:,:,deriv(1),deriv(2),deriv(3)) = 0.0_dp
            ierr = add_arr_mult(theta_s(:,:,:,0:,1:,0:),&
                &eq_1%flux_t_E(:,1:)/(2*pi),&
                &eq_2%det_T_EF(:,:,:,deriv(1),deriv(2),deriv(3)),deriv)
            chckerr('')
        end if
    end function calc_t_vf_ind
    !> \private array version
    integer function calc_t_vf_arr(grid_eq,eq_1,eq_2,deriv) result(ierr)
        character(*), parameter :: rout_name = 'calc_T_VF_arr'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                             !< equilibrium grid
        type(eq_1_type), intent(in) :: eq_1                                !< flux equilibrium
        type(eq_2_type), intent(inout) :: eq_2                             !< metric equilibrium
        integer, intent(in) :: deriv(:,:)                                  !< derivatives

        ! local variables
        integer :: id
```

```fortran

        ! initialize ierr
        ierr = 0

        do id = 1, size(deriv,2)
            ierr = calc_t_vf_ind(grid_eq,eq_1,eq_2,deriv(:,id))
            chckerr('')
        end do
    end function calc_t_vf_arr

    !> \private individual version
    integer function calc_t_hf_ind(grid_eq,eq_1,eq_2,deriv) result(ierr)
        use num_vars, only: use_pol_flux_f
        use num_utilities, only: add_arr_mult, c

        character(*), parameter :: rout_name = 'calc_T_HF_ind'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(eq_1_type), intent(in) :: eq_1                                     !< flux equilibrium
        type(eq_2_type), intent(inout) :: eq_2                                  !< metric equilibrium
        integer, intent(in) :: deriv(:)                                         !< derivatives

        ! local variables
        integer :: kd                                                           ! counter
        real(dp), allocatable :: theta_s(:,:,:,:,:,:)                           ! theta_F and derivatives
        real(dp), allocatable :: zeta_s(:,:,:,:,:,:)                            ! - zeta_F and derivatives
        integer :: dims(3)                                                      ! dimensions

        ! initialize ierr
        ierr = 0

#if ldebug
        ! check the derivatives requested
        ierr = check_deriv(deriv,max_deriv,'calc_T_HF')
        chckerr('')
#endif

        ! set up dims
        dims = [grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r]

        ! initialize
        eq_2%T_EF(:,:,:,:,deriv(1),deriv(2),deriv(3)) = 0._dp

        ! set up theta_s
        allocate(theta_s(dims(1),dims(2),dims(3),&
            &0:deriv(1)+1,0:deriv(2)+1,0:deriv(3)+1))
        theta_s = 0.0_dp
        theta_s(:,:,:,0,0,0) = grid_eq%theta_E
        theta_s(:,:,:,0,1,0) = 1.0_dp

        if (use_pol_flux_f) then
            ! calculate transformation matrix T_H^F
            ! (1,1)
            ierr = add_arr_mult(theta_s,-eq_1%q_saf_E(:,1:),&
                &eq_2%T_EF(:,:,:,c([1,1],.false.),deriv(1),deriv(2),deriv(3)),&
                &deriv)
            chckerr('')
            ! (1,2)
            if (sum(deriv).eq.0) then
                eq_2%T_EF(:,:,:,c([1,2],.false.),0,0,0) = 1._dp
            !else
                !eq_2%T_EF(:,:,:,c([1,2],.false.),deriv(1),deriv(2),deriv(3)) &
                    !&= 0.0_dp
            end if
            ! (1,3)
            !eq_2%T_EF(:,:,:,c([1,3],.false.),deriv(1),deriv(2),deriv(3)) = &
                !&0.0_dp
            ! (2,1)
            if (deriv(2).eq.0 .and. deriv(3).eq.0) then
                do kd = 1,dims(3)
                    eq_2%T_EF(:,:,kd,c([2,1],.false.),deriv(1),0,0) = &
                        &-eq_1%q_saf_E(kd,deriv(1))
                end do
            !else
                !eq_2%T_EF(:,:,:,c([2,1],.false.),deriv(1),deriv(2),deriv(3)) &
                    !&= 0.0_dp
            end if
            ! (2,2)
            !eq_2%T_EF(:,:,:,c([2,2],.false.),deriv(1),deriv(2),deriv(3)) = &
                !&0.0_dp
            ! (2,3)
            if (sum(deriv).eq.0) then
                eq_2%T_EF(:,:,:,c([2,3],.false.),0,0,0) = 1.0_dp
            !else
                !eq_2%T_EF(:,:,:,c([2,3],.false.),deriv(1),deriv(2),deriv(3)) &
                    !&= 0.0_dp
```

```
            end if
            ! (3,1)
            if (sum(deriv).eq.0) then
                eq_2%T_EF(:,:,:,c([3,1],.false.),0,0,0) = 1.0_dp
            !else
                !eq_2%T_EF(:,:,:,c([3,1],.false.),deriv(1),deriv(2),deriv(3)) &
                    !&= 0.0_dp
            end if
            ! (3,2)
            !eq_2%T_EF(:,:,:,c([3,2],.false),deriv(1),deriv(2),deriv(3)) = &
                !&0.0_dp
            ! (3,3)
            !eq_2%T_EF(:,:,:,c([3,3],.false),deriv(1),deriv(2),deriv(3)) = &
                !&0.0_dp

            ! determinant
            eq_2%det_T_EF(:,:,:,deriv(1),deriv(2),deriv(3)) = 0.0_dp
            if (sum(deriv).eq.0) then
                eq_2%det_T_EF(:,:,:,deriv(1),0,0) = 1._dp
            !else
                !eq_2%det_T_EF(:,:,deriv(1),deriv(2),deriv(3)) = 0.0_dp
            end if
        else
            ! set up zeta_s
            allocate(zeta_s(dims(1),dims(2),dims(3),&
                &0:deriv(1)+1,0:deriv(2)+1,0:deriv(3)+1))
            zeta_s = 0.0_dp
            ! start from zeta_E
            zeta_s(:,:,:,0,0,0) = grid_eq%zeta_E
            zeta_s(:,:,:,0,0,1) = 1.0_dp

            ! calculate transformation matrix T_H^F
            ! (1,1)
            ierr = add_arr_mult(zeta_s,eq_1%rot_t_E(:,1:),&
                &eq_2%T_EF(:,:,:,c([1,1],.false.),deriv(1),deriv(2),deriv(3)),&
                &deriv)
            chckerr('')
            ! (1,2)
            if (deriv(2).eq.0 .and. deriv(3).eq.0) then
                do kd = 1,dims(3)
                    eq_2%T_EF(:,:,kd,c([1,2],.false.),deriv(1),0,0) = &
                        &eq_1%q_saf_E(kd,deriv(1))
                end do
            !else
                !eq_2%T_EF(:,:,:,c([1,2],.false.),deriv(1),deriv(2),deriv(3)) &
                    !&= 0.0_dp
            end if
            ! (1,3)
            !eq_2%T_EF(:,:,:,c([1,3],.false.),deriv(1),deriv(2),deriv(3)) = &
                !&0.0_dp
            ! (2,1)
            if (sum(deriv).eq.0) then
                eq_2%T_EF(:,:,:,c([2,1],.false.),0,0,0) = -1.0_dp
            !else
                !eq_2%T_EF(:,:,:,c([2,1],.false.),deriv(1),deriv(2),deriv(3)) &
                    !&= 0.0_dp
            end if
            ! (2,2)
            !eq_2%T_EF(:,:,:,c([2,2],.false.),deriv(1),deriv(2),deriv(3)) = &
                !&0.0_dp
            ! (2,3)
            !eq_2%T_EF(:,:,:,c([2,3],.false.),deriv(1),deriv(2),deriv(3)) = &
                !&0.0_dp
            ! (3,1)
            if (deriv(2).eq.0 .and. deriv(3).eq.0) then
                do kd = 1,dims(3)
                    eq_2%T_EF(:,:,kd,c([3,1],.false.),deriv(1),0,0) = &
                        &eq_1%rot_t_E(kd,deriv(1))
                end do
            !else
                !eq_2%T_EF(:,:,:,c([3,1],.false.),deriv(1),deriv(2),deriv(3)) &
                    !&= 0.0_dp
            end if
            ! (3,2)
            !eq_2%T_EF(:,:,:,c([3,2],.false.),deriv(1),deriv(2),deriv(3)) = &
                !&0.0_dp
            ! (3,3)
            if (sum(deriv).eq.0) then
                eq_2%T_EF(:,:,:,c([3,3],.false.),deriv(1),0,0) = 1.0_dp
            !else
                !eq_2%T_EF(:,:,:,c([3,3],.false.),deriv(1),deriv(2),deriv(3)) &
                    !&= 0.0_dp
            end if

            ! determinant
            eq_2%det_T_EF(:,:,:,deriv(1),deriv(2),deriv(3)) = 0.0_dp
            if (deriv(2).eq.0 .and. deriv(3).eq.0) then
```

```
                do kd = 1,dims(3)
                    eq_2%det_T_EF(:,:,kd,deriv(1),0,0) = &
                        &eq_1%q_saf_E(kd,deriv(1))
                end do
            !else
                !eq_2%det_T_EF(:,:,deriv(1),deriv(2),deriv(3)) = 0.0_dp
            end if
        end if
    end function calc_t_hf_ind
    !> \private array version
    integer function calc_t_hf_arr(grid_eq,eq_1,eq_2,deriv) result(ierr)
        character(*), parameter :: rout_name = 'calc_T_HF_arr'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(eq_1_type), intent(in) :: eq_1                                     !< flux equilibrium
        type(eq_2_type), intent(inout) :: eq_2                                  !< metric equilibrium
        integer, intent(in) :: deriv(:,:)                                       !< derivatives

        ! local variables
        integer :: id

        ! initialize ierr
        ierr = 0

        do id = 1, size(deriv,2)
            ierr = calc_t_hf_ind(grid_eq,eq_1,eq_2,deriv(:,id))
            chckerr('')
        end do
    end function calc_t_hf_arr

    !> Plots the flux quantities in the solution grid.
    !!
    !!  - safety factor \c q_saf
    !!  - rotational transform \c rot_t
    !!  - pressure \c pres
    !!  - poloidal flux \c flux_p
    !!  - toroidal flux \c flux_t
    !!
    !! \return ierr
    integer function flux_q_plot(grid_eq,eq) result(ierr)
        use num_vars, only: rank, no_plots, use_normalization
        use grid_utilities, only: trim_grid
        use mpi_utilities, only: get_ser_var
        use eq_vars, only: pres_0, psi_0

        character(*), parameter :: rout_name = 'flux_q_plot'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                                  !< normal grid
        type(eq_1_type), intent(in) :: eq                                       !< flux equilibrium variables

        ! local variables
        integer :: norm_id(2)                                                   ! untrimmed normal indices for trimmed grids
        integer :: id                                                           ! counter
        integer :: n_vars = 5                                                   ! nr. of variables to plot
        character(len=max_str_ln), allocatable :: plot_titles(:)                ! plot titles
        type(grid_type) :: grid_trim                                            ! trimmed grid
        real(dp), allocatable :: x_plot_2d(:,:)                                 ! x values of 2D plot
        real(dp), allocatable :: y_plot_2d(:,:)                                 ! y values of 2D plot

        ! initialize ierr
        ierr = 0

        ! bypass plots if no_plots
        if (no_plots) return

        ! user output
        call writo('Plotting flux quantities')

        call lvl_ud(1)

        ! trim grid
        ierr = trim_grid(grid_eq,grid_trim,norm_id)
        chckerr('')

        ! initialize plot titles and file names
        allocate(plot_titles(n_vars))
        plot_titles(1) = 'safety factor []'
        plot_titles(2) = 'rotational transform []'
        plot_titles(3) = 'pressure [pa]'
        plot_titles(4) = 'poloidal flux [Tm^2]'
        plot_titles(5) = 'toroidal flux [Tm^2]'

        ! plot using HDF5
        ierr = flux_q_plot_hdf5()
        chckerr('')
```

```

        ! plot using external program
        ierr = flux_q_plot_ex()
        chckerr('')

        ! clean up
        call grid_trim%dealloc()

        call lvl_ud(-1)
    contains
        ! plots the flux quantities in HDF5
        !> \private
        integer function flux_q_plot_hdf5() result(ierr)
            use num_vars, only: eq_style
            use output_ops, only: plot_hdf5
            use grid_utilities, only: extend_grid_f, trim_grid, calc_xyz_grid
            use vmec_utilities, only: calc_trigon_factors

            character(*), parameter :: rout_name = 'flux_q_plot_HDF5'

            ! local variables
            integer :: kd                                                   ! counter
            real(dp), allocatable :: x_plot(:,:,:,:)                        ! x values of total plot
            real(dp), allocatable :: y_plot(:,:,:,:)                        ! y values of total plot
            real(dp), allocatable :: z_plot(:,:,:,:)                        ! z values of total plot
            real(dp), allocatable :: f_plot(:,:,:,:)                        ! values of variable of total plot
            integer :: plot_dim(4)                                          ! total plot dimensions
            integer :: plot_offset(4)                                       ! plot offset
            type(grid_type) :: grid_plot                                    ! grid for plotting
            character(len=max_str_ln) :: file_name                          ! file name

            ! initialize ierr
            ierr = 0

            ! set up file name
            file_name = 'flux_quantities'

            ! fill the 2D version of the plot
            allocate(y_plot_2d(grid_trim%loc_n_r,n_vars))

            y_plot_2d(:,1) = eq%q_saf_FD(norm_id(1):norm_id(2),0)
            y_plot_2d(:,2) = eq%rot_t_FD(norm_id(1):norm_id(2),0)
            y_plot_2d(:,3) = eq%pres_FD(norm_id(1):norm_id(2),0)
            y_plot_2d(:,4) = eq%flux_p_FD(norm_id(1):norm_id(2),0)
            y_plot_2d(:,5) = eq%flux_t_FD(norm_id(1):norm_id(2),0)

            ! extend trimmed equilibrium grid
            ierr = extend_grid_f(grid_trim,grid_plot,grid_eq=grid_eq)
            chckerr('')

            ! if VMEC, calculate trigonometric factors of plot grid
            if (eq_style.eq.1) then
                ierr = calc_trigon_factors(grid_plot%theta_E,grid_plot%zeta_E,&
                    &grid_plot%trigon_factors)
                chckerr('')
            end if

            ! set up plot_dim and plot_offset
            plot_dim = [grid_plot%n(1),grid_plot%n(2),grid_plot%n(3),n_vars]
            plot_offset = [0,0,grid_plot%i_min-1,n_vars]

            ! set up total plot variables
            allocate(x_plot(grid_plot%n(1),grid_plot%n(2),grid_plot%loc_n_r,&
                &n_vars))
            allocate(y_plot(grid_plot%n(1),grid_plot%n(2),grid_plot%loc_n_r,&
                &n_vars))
            allocate(z_plot(grid_plot%n(1),grid_plot%n(2),grid_plot%loc_n_r,&
                &n_vars))
            allocate(f_plot(grid_plot%n(1),grid_plot%n(2),grid_plot%loc_n_r,&
                &n_vars))

            ! calculate 3D X,Y and Z
            ierr = calc_xyz_grid(grid_eq,grid_plot,x_plot(:,:,:,1),&
                &y_plot(:,:,:,1),z_plot(:,:,:,1))
            chckerr('')
            do id = 2,n_vars
                x_plot(:,:,:,id) = x_plot(:,:,:,1)
                y_plot(:,:,:,id) = y_plot(:,:,:,1)
                z_plot(:,:,:,id) = z_plot(:,:,:,1)
            end do

            do kd = 1,grid_plot%loc_n_r
                f_plot(:,:,kd,1) = y_plot_2d(kd,1)                          ! safey factor
                f_plot(:,:,kd,2) = y_plot_2d(kd,2)                          ! rotational transform
                f_plot(:,:,kd,3) = y_plot_2d(kd,3)                          ! pressure
                f_plot(:,:,kd,4) = y_plot_2d(kd,4)                          ! poloidal flux
                f_plot(:,:,kd,5) = y_plot_2d(kd,5)                          ! toroidal flux
```

```
            end do

            ! rescale if normalized
            if (use_normalization) then
                f_plot(:,:,:,3) = f_plot(:,:,:,3)*pres_0                        ! pressure
                f_plot(:,:,:,4) = f_plot(:,:,:,4)*psi_0                         ! flux_p
                f_plot(:,:,:,5) = f_plot(:,:,:,5)*psi_0                         ! flux_t
            end if

            ! print the output using HDF5
            call plot_hdf5(plot_titles,file_name,f_plot,plot_dim,plot_offset,&
                &x_plot,y_plot,z_plot,col=1,descr='Flux quantities')

            ! deallocate and destroy grid
            deallocate(y_plot_2d)
            deallocate(x_plot,y_plot,z_plot,f_plot)
            call grid_plot%dealloc()
        end function flux_q_plot_hdf5

        ! plots the pressure and fluxes in external program
        !> \private
        integer function flux_q_plot_ex() result(ierr)
            use eq_vars, only: max_flux_f

            character(*), parameter :: rout_name = 'flux_q_plot_ex'

            ! local variables
            character(len=max_str_ln), allocatable :: file_name(:)              ! file_name
            real(dp), allocatable :: ser_var_loc(:)                             ! local serial var

            ! initialize ierr
            ierr = 0

            ! allocate variabels if master
            if (rank.eq.0) then
                allocate(x_plot_2d(grid_trim%n(3),n_vars))
                allocate(y_plot_2d(grid_trim%n(3),n_vars))
            end if

            ! fill the 2D version of the plot
            !ierr = get_ser_var(eq%q_saf_FD(norm_id(1):norm_id(2),0),ser_var_loc)
            !CHCKERR('')
            !if (rank.eq.0) Y_plot_2D(:,1) = ser_var_loc
            !ierr = get_ser_var(eq%rot_t_FD(norm_id(1):norm_id(2),0),ser_var_loc)
            !CHCKERR('')
            !if (rank.eq.0) Y_plot_2D(:,2) = ser_var_loc
            ierr = get_ser_var(eq%pres_FD(norm_id(1):norm_id(2),0),ser_var_loc)
            chckerr('')
            if (rank.eq.0) y_plot_2d(:,3) = ser_var_loc
            ierr = get_ser_var(eq%flux_p_FD(norm_id(1):norm_id(2),0),&
                &ser_var_loc)
            chckerr('')
            if (rank.eq.0) y_plot_2d(:,4) = ser_var_loc
            ierr = get_ser_var(eq%flux_t_FD(norm_id(1):norm_id(2),0),&
                &ser_var_loc)
            chckerr('')
            if (rank.eq.0) y_plot_2d(:,5) = ser_var_loc

            ! rescale if normalized
            if (use_normalization .and. rank.eq.0) then
                y_plot_2d(:,3) = y_plot_2d(:,3)*pres_0                          ! pressure
                y_plot_2d(:,4) = y_plot_2d(:,4)*psi_0                           ! flux_p
                y_plot_2d(:,5) = y_plot_2d(:,5)*psi_0                           ! flux_t
            end if

            ! continue the plot if master
            if (rank.eq.0) then
                ! deallocate local serial variables
                deallocate(ser_var_loc)

                ! set up file names
                allocate(file_name(2))
                file_name(1) = 'pres'
                file_name(2) = 'flux'

                ! 2D normal variable (normalized F coordinate)
                x_plot_2d(:,1) = grid_trim%r_F*2*pi/max_flux_f
                do id = 2,n_vars
                    x_plot_2d(:,id) = x_plot_2d(:,1)
                end do

                ! plot the  individual 2D output  of this process  (except q_saf
                ! and rot_t, as they are already in plot_resonance)
                call print_ex_2d(plot_titles(3),file_name(1),&
                    &y_plot_2d(:,3),x_plot_2d(:,3),draw=.false.)
                ! fluxes
                call print_ex_2d(plot_titles(4:5),file_name(2),&
```

```
                    &y_plot_2d(:,4:5),x_plot_2d(:,4:5),draw=.false.)

                ! draw plot
                call draw_ex(plot_titles(3:3),file_name(1),1,1,.false.)         ! pressure
                call draw_ex(plot_titles(4:5),file_name(2),2,1,.false.)         ! fluxes

                ! user output
                call writo('Safety factor and rotational transform are plotted &
                    &using "plot_resonance"')

                ! clean up
                deallocate(x_plot_2d,y_plot_2d)
            end if
        end function flux_q_plot_ex
    end function flux_q_plot

    !> Calculates derived equilibrium quantities system.
    !!
    !!  - magnetic shear \c S
    !!  - normal curvature \c kappa_n
    !!  - geodesic curvature \c kappa_g
    !!  - parallel current \c sigma
    !!
    !! Naive implementations  of these quantities,  with the exception of  \c S,
    !! give numerically very unfavorable results,  so special care must be taken
    !! in this procedure.
    !!
    !! This is  an important  issue as these  derived equilbrium  quantities are
    !! important building  blocks of  the perturbed potential  energy, including
    !! the drivers of instabilities.
    !!
    !! The  general formulas  are derived  under  the first  section, VMEC.  For
    !! axisymmetric HELENA, however, simplified equations are possible and these
    !! are presented afterwards.
    !!
    !! VMEC
    !! ====
    !!
    !! The shear
    !! ---------
    !!
    !! The  local shear  \f$S\f$ is  calculated using  equation 3.22  from \cite
    !! Weyens3D :
    !!
    !! \f$S = - \frac{1}{\mathcal{J}} \frac{\partial}{\partial \theta}
    !!  \left(\frac{ h^{\psi\alpha}}{ h^{\psi \psi}}\right)\f$
    !!
    !! which  doesn't pose  any particular  problems as  there are  only angular
    !! derivatives.
    !!
    !! The curvature
    !! -------------
    !!
    !! This is  calculated using  the parallel derivative  of the  parallel unit
    !! vector (i.e.  theta if  using poloidal  flux and  zeta if  using toroidal
    !! flux).
    !!
    !! By  taking instead  the  derivative  of the  covariant  basis vector  and
    !! realizing that  the difference with the  real curvature lies solely  in a
    !! component along the parallel direction, which cancels out when taking the
    !! normal or geodesic components, the situation becomes easier.
    !!
    !! Asuming  the poloidal  flux is  used as  normal coordinate,  the taks  is
    !! therefore how to calculate
    !!  \f$\frac{1}{\left|\vec{e}_\theta\right|^2} \frac{\partial}{\partial \theta} \vec{e}_\theta\f$.
    !!
    !! By  transforming both  the  derivative as  well as  the  basis vector  to
    !! Equilibrium coordinates,  this can be  written as
    !!
    !!  \f$\sum_{i=2,3} \sum_{j=2,3} \mathcal{T}_\text{F}^\text{E} \left(3,i\right))
    !!      \frac{\partial}{\partial u^i_\text{E}} \left(
    !!      \mathcal{T}_\text{F}^\text{E} \left(3,j\right) \vec{e}_{j,\text{E}}\right)\f$
    !!
    !! where   the   summations   only   run  over   the   angular   coordinates
    !! because    \f$\vec{e}_\theta\f$   never    has   any    component   along
    !! \f$\vec{e}_{\psi,\text{E}}\f$.
    !!
    !! This double summation formula can then be written as a matrix equation
    !!
    !!  \f$\begin{pmatrix}\mathcal{T}_\text{F}^\text{E}\left(3,2\right) &
    !!      \mathcal{T}_\text{F}^\text{E}\left(3,3\right)\end{pmatrix}
    !!      \left[
    !!      \begin{pmatrix} \frac{\partial}{\partial u^2} \vec{e}_{2} &
    !!          \frac{\partial}{\partial u^2} \vec{e}_{3} \\
    !!          \frac{\partial}{\partial u^3} \vec{e}_{2} &
    !!          \frac{\partial}{\partial u^3} \vec{e}_{3}\end{pmatrix}_\text{E}
    !!      \begin{pmatrix}\mathcal{T}_\text{F}^\text{E}\left(3,2\right) \\
```

```
!!      \mathcal{T}_\text{F}^\text{E}\left(3,3\right)\end{pmatrix}
!!      +
!!      \begin{pmatrix}
!!          \frac{\partial}{\partial u^2} \mathcal{T}_\text{F}^\text{E}\left(3,2\right) &
!!          \frac{\partial}{\partial u^2} \mathcal{T}_\text{F}^\text{E}\left(3,3\right) \\
!!          \frac{\partial}{\partial u^3} \mathcal{T}_\text{F}^\text{E}\left(3,2\right) &
!!          \frac{\partial}{\partial u^3} \mathcal{T}_\text{F}^\text{E}\left(3,3\right)
!!      \end{pmatrix}
!!      \begin{pmatrix}\vec{e}_2 \\ \vec{e}_3\end{pmatrix}
!!      \right]\f$
!!
!! which can be represented shorthand as
!!  \f$\mathcal{T}_\text{F}^\text{E}\left(3,2:3\right) \left[
!!      \mathcal{D}\vec{e}_\text{E} \mathcal{T}_\text{F}^\text{E}\left(3,2:3\right)^T
!!      +\mathcal{D} \mathcal{T} \vec{e}_\text{E}^T \right]\f$
!!
!! It     is     relatively     easy     to     set     up     the     matrix
!! \f$\mathcal{D}\vec{e}_\text{E}\f$  as  a   function  of  covariant  basis
!! vectors in the Cylindrical coordinate system.
!! The  derivatives of  the  transformation matrix  itself  can likewise  be
!! found.
!!
!! The steps used in this routine are therefore
!!  -# Set up  \f$\mathcal{D}\vec{e}_\text{E}\f$, i.e. as a  function of the
!!  three cylindrical covariant basis vectors.
!!  -# Set up  correctcion by the derivatives of  the transformation matrix,
!!  with a single summation.
!!  -# Decompose the normal \f$\frac{\nabla \psi}{\left|\nabla\psi\right|^2}\f$
!!  and geodesic  vectors \f$\frac{\nabla \psi \times  \vec{B}}{B^2}\f$ as a
!!  function of the cylindrical contravariant basis vectors.
!!  -# Double summation to reduce term ~ \f$\mathcal{D}\vec{e}_\text{E}\f$
!!  and correction by single summation to reduce term ~
!!  \f$\mathcal{D}\mathcal{T}\f$.
!!  -# Dot these  for each of the four (actually  three because of symmetry)
!!  elements of \f$\mathcal{D}\vec{e}_\text{E}\f$.
!!  -# Divide by \f$\left|\vec{e}_\theta\right|^2\f$.
!!  -# Possibly correct for toroidal flux.
!!
!! An advantage of using this formulation  is that no normal derivatives are
!! needed, so that nothing has to implicitely cancel out.
!!
!! The parallel current
!! --------------------
!!
!! The  parallel current  is  calculated from  the shear  with  the help  of
!! equation 3.29 of \cite Weyens3D :
!!
!! \f$\mu_0 \sigma = -\frac{1}{B_\theta} \left[
!!  2 \frac{\nabla \psi \times \vec{B}}{\left|\nabla \psi\right|^2} \cdot
!!  \frac{\partial \left(\nabla \psi\right)}{\partial \theta} +
!!  \mathcal{J} \left|\nabla \psi\right|^2 S \right]\f$ .
!!
!! where a  similar technique can be  used as above, for  the calculation of
!! the curvature: As
!!
!! \f$\frac{\nabla \psi \times \vec{B}}{\left|\nabla \psi\right|^2}
!!  = \frac{B_\theta \vec{e}_\alpha - B_\alpha \vec{e}_\theta}
!!  {\mathcal{J} \left|\nabla\psi\right|^2}\f$ ,
!!
!! The parallel derivative of \f$\nabla \psi\f$ can be rewritten in terms of
!! contravariant components of derivatives of covariant basis vectors:
!!
!! \f$\left\{ \begin{aligned}
!!  \vec{e}_\alpha  \cdot \frac{\partial \left(\nabla \psi\right)}{\partial\theta}
!!  = - \nabla \psi \cdot \frac{\partial \vec{e}_\theta}{\partial \alpha} \\
!!  \vec{e}_\theta  \cdot \frac{\partial \left(\nabla \psi\right)}{\partial\theta}
!!  = - \nabla \psi \cdot \frac{\partial \vec{e}_\theta}{\partial \theta} \\
!! \end{aligned}\right. \f$ ,
!!
!! so that the result is:
!!
!! \f$\frac{\nabla \psi \times \vec{B}}{\left|\nabla \psi\right|^2} \cdot
!!  \frac{\partial \left(\nabla \psi\right)}{\partial \theta}
!!  = \frac{1}{\mathcal{J}^2} \frac{\nabla \psi}{\left|\nabla\psi\right|^2} \cdot
!!  \left[ g_{\alpha\theta} \frac{\partial \vec{e}_\theta}{\partial \theta}
!!      - g_{\theta\theta} \frac{\partial \vec{e}_\theta}{\partial \alpha} \right]
!! \f$ ,
!!
!! which  is given  by a  formula  similar to  the  one used  above for  the
!! geodesical curvature.
!!
!! In  debug  mode,  it  can  be  checked  whether  the  current  is  indeed
!! divergence-free, with the help of equation 3.33 of \cite Weyens3D.
!!
!! \f$ -2 p' \int_{\theta_0}^\theta \mathcal{J} \kappa_g \text{d}{\theta} =
!!      \sigma\left(\theta\right) - \sigma_0\f$
!!
```

```
!! and whether a  direct, naive implementation of the  parallel current from
!! equation 3.26  of \cite  Weyens3D agrees with  the more  accurate results
!! used here:
!!
!! \f$\mu_0 \sigma = \frac{\partial B_\psi}{\partial \alpha} -
!!      \frac{\partial B_\alpha}{\partial \psi} -
!!      \mu_0 p' \mathcal{J} \frac{B_\alpha}{B_\theta}\f$ .
!!
!! The reason why they are  generally different is that this implementation
!! relies on the cancellation of large terms.
!!
!! HELENA
!! ======
!!
!! The parallel current
!! --------------------
!!
!! As \f$B_\alpha = F\left(\psi\right)\f$ and
!! \f$\vec{e}_{\alpha,\text{F}}  =   \vec{e}_{\phi,\text{H}}\f$,  the  naive
!! expression for the shear becomes simply
!!
!! \f$\sigma = -F' - \mu_0 p' \frac{F}{B^2}\f$ ,
!!
!! which can be used like that.
!!
!!
!! The shear
!! ---------
!!
!! The calculate  the local shear  \f$S\f$ is looks like  it is best  to use
!! equation 3.22 from \cite Weyens3D , just as in the VMEC case:
!!
!! As a test, however, equation 3.29 of \cite Weyens3D can be used in stead:
!!
!! \f$\mathcal{J}\left|\nabla \psi\right|^2 S + \mu_0 \mathcal{J}B^2 \sigma
!! = - 2 \frac{F}{R} \left( \frac{Z_\theta}{R} +
!! \frac{Z_\theta R_{\theta\theta} - R_\theta Z_{\theta\theta}}{R_\theta^2 + Z_\theta^2} \right)\f$.
!!
!! from which \f$\sigma\f$ can be obtained.
!!
!! The curvature
!! -------------
!!
!! Also  the  curvature  expressions  can  be  simplified  for  axisymmetric
!! situations. The result is given by
!!
!! \f$\kappa_n = \frac{q R}{F}
!!  \frac{Z_\theta \left( R_{\theta\theta} - q^2 R\right) - R_\theta Z_{\theta\theta}}
!!  {\left(R_\theta^2 + Z_\theta^2 + \left(q R\right)^2\right) \left( R_\theta^2 + Z_\theta^2 \right)} \f$
!!
!! \f$\kappa_g = q R
!!  \frac{R_\theta \left( 2 \left(R_\theta^2 + Z_\theta^2\right) + \left(qR\right)^2 \right)
!!  - R \left(R_\theta R_{\theta\theta} + Z_\theta Z_{\theta\theta}\right)}
!!  {\left(R_\theta^2 + Z_\theta^2 + \left(q R\right)^2\right)^2} \f$
!!
!!
!! \note
!!  -# If the toroidal flux is  used instead, the actual curvature obviously
!!  is  unchanged,  which  implies  that  the normal  component  has  to  be
!!  multiplied by  the safety factor  and the  geodesic component has  to be
!!  divided by it.
!!  -# The formulas for the normal and geodesic basis vectors for VMEC are
!!      * \f$\frac{\nabla \psi}{\left|\nabla\psi\right|^2} =
!!          -\frac{\mathcal{J} \left(1 + \lambda_\theta\right)}
!!          {Z_\theta^2 + R_\theta^2 + \left(R_\zeta Z_\theta - R_\theta Z_\zeta \right)/R^2} \frac{1}{R}
!!          \left( -Z_\theta , R_\zeta Z_\theta - R_\theta Z_\zeta, R_\theta\right)_\text{C}\f$
!!      * \f$\frac{\nabla \psi \times  \vec{B}}{B^2} =
!!          \frac{1}{g_{\theta\theta}}
!!          \left(P R_\theta - Q R_\zeta, -Q R^2, P Z_\theta - Q Z_\zeta\right)_\text{C}\f$
!!      *\f$Q = g_{\theta\theta} - q g_{\theta\alpha}\f$ (using right-handed
!!      flux \f$q_\text{F}\f$, which is the  inverse of the left-handed VMEC
!!      \f$q_\text{V}\f$)
!!      *\f$P = \frac{Q \lambda_\zeta - g_{\alpha\theta}}{1+\lambda_\theta}\f$
!!      (where the derivatives of \f$R\f$,  \f$Z\f$ and \f$\lambda\f$ are in
!!      VMEC coordinates).
!!  -# The formulas for the normal and geodesic basis vectors for HELENA are
!!      * \f$\frac{\nabla \psi}{\left|\nabla\psi\right|^2} =
!!          -\frac{1}{Z_\theta^2 + R_\theta^2} \frac{q R}{F}
!!          \left( -Z_\theta , 0, R_\theta\right)_\text{C}\f$
!!      * \f$\frac{\nabla \psi \times  \vec{B}}{B^2} =
!!          \frac{q R^2}{R_\theta^2 + Z_\theta^2 + q^2 R^2}
!!          \left(-R_\theta, -\frac{R_\theta^2 + Z_\theta^2}{q}, -Z_\theta\right)_\text{C}\f$
integer function calc_derived_q(grid_eq,eq_1,eq_2) result(ierr)
    use eq_vars, only: vac_perm, max_flux_f
    use num_vars, only: eq_style, use_pol_flux_f
    use helena_vars, only: r_h, z_h, chi_h, ias
    use num_utilities, only: spline
```

```fortran

        character(*), parameter :: rout_name = 'calc_derived_q'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(eq_1_type), intent(in) :: eq_1                                     !< flux equilibrium variables
        type(eq_2_type), intent(inout), target :: eq_2                          !< metric equilibrium variables

        ! local variables
        integer :: id, jd, kd, ld                                               ! counters
        integer :: kd_h                                                         ! kd in Helena tables
        integer :: bcs(2,2)                                                     ! boundary conditions (theta(even), theta(odd))
        real(dp) :: bcs_val(2,2)                                                ! values for boundary conditions
        real(dp), allocatable :: d1_epar(:,:,:,:)                               ! cylindrical contravariant components of alpha derivative of parallel basis vector
        real(dp), allocatable :: d3_epar(:,:,:,:)                               ! cylindrical contravariant components of theta derivative of parallel basis vector
        real(dp), allocatable :: b_n(:,:,:,:)                                   ! covariant Cylindrical components of normal basis vector
        real(dp), allocatable :: b_g(:,:,:,:)                                   ! covariant Cylindrical components of geodesic basis vector
        real(dp), allocatable :: de(:,:,:,:,:,:)                                ! derivs of cov. unit vector in E (space,deriv.,unitvec,output)
        real(dp), allocatable :: d_de(:,:,:,:,:)                                ! derivs of transf. matrix in E (space,deriv.,output)
        real(dp), allocatable :: rchi(:,:,:)                                    ! chi and chi^2 derivatives of R
        real(dp), allocatable :: zchi(:,:,:)                                    ! chi and chi^2 derivatives of Z

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Set up derived quantities in flux coordinates')

        call lvl_ud(1)

        ! initialize helper variables
        allocate(de(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r,2,2,3))
        allocate(d_de(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r,2,3))
        allocate(b_n(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r,3))
        allocate(b_g(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r,3))
        allocate(d1_epar(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r,3))
        allocate(d3_epar(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r,3))

        select case (eq_style)
            case (1)                                                            ! VMEC
                ! Calculate the shear S
                call calc_derived_s_direct(eq_2,eq_2%S)

                ! Equilibrium contravariant components of angular derivatives of
                ! covariant parallel basis vector
                call calc_derived_de_epar_vmec(grid_eq,eq_2,de,d_de,b_n,b_g)
            case (2)                                                            ! HELENA
                ! Calculate parallel current sigma
                call calc_derived_sigma_hel(grid_eq,eq_2,eq_2%sigma)

                ! set up boundary conditions for theta derivatives
                if (ias.eq.0) then                                              ! top-bottom symmetric
                    bcs(:,1) = [1,1]                                            ! theta(even): zero first derivative
                    bcs(:,2) = [2,2]                                            ! theta(odd): zero first derivative
                else
                    bcs(:,1) = [-1,-1]                                          ! theta(even): periodic
                    bcs(:,2) = [-1,-1]                                          ! theta(odd): periodic
                end if
                bcs_val = 0._dp

                ! initialize output
                de = 0._dp
                d_de = 0._dp
                b_n = 0._dp
                b_g = 0._dp

                ! calculate the derivatives of covariant unit vectors in
                ! cEquilibrium oordinates
                allocate(rchi(grid_eq%n(1),grid_eq%loc_n_r,0:2))
                allocate(zchi(grid_eq%n(1),grid_eq%loc_n_r,0:2))
                rchi(:,:,0) = r_h(:,grid_eq%i_min:grid_eq%i_max)
                zchi(:,:,0) = z_h(:,grid_eq%i_min:grid_eq%i_max)
                do kd = 1,grid_eq%loc_n_r
                    kd_h = grid_eq%i_min-1+kd
                    do id = 1,2
                        ierr = spline(chi_h,r_h(:,kd_h),chi_h,rchi(:,kd,id),&
                            &ord=3,deriv=id,bcs=bcs(:,1),bcs_val=bcs_val(:,1))  ! even
                        chckerr('')
                        ierr = spline(chi_h,z_h(:,kd_h),chi_h,zchi(:,kd,id),&
                            &ord=3,deriv=id,bcs=bcs(:,2),bcs_val=bcs_val(:,2))  ! odd
```

```
                        chckerr('')
                    end do
                end do

                ! Equilibrium contravariant components of angular derivatives of
                ! covariant parallel basis vector
                call calc_derived_de_epar_hel(grid_eq,eq_1,rchi,zchi,&
                    &de,d_de,b_n,b_g)
        end select

        ! calculate cylindrical contravariant  components of angular derivatives
        ! of covariant parallel basis vector
        call calc_derived_dc_epar(de,d_de,eq_2%T_FE(:,:,:,:,0,0,0),&
            &d1_epar,d3_epar)

        ! clean up
        deallocate(de,d_de)

        ! calculate normal and geodesic curvature
        call calc_derived_kappa_from_e(grid_eq,eq_1,eq_2,b_n,b_g,&
            eq_2%kappa_n,eq_2%kappa_g)

        select case (eq_style)
            case (1)                                                    ! VMEC
                ! calculate parallel current
                call calc_derived_sigma_from_e(eq_2,b_n,eq_2%sigma)
            case (2)                                                    ! HELENA
                ! Calculate the shear S
                ierr = calc_derived_s_from_deriv_hel(grid_eq,eq_2,bcs(:,1),&
                    &bcs_val(:,1),eq_2%S)                               ! even
                chckerr('')
        end select

        ! clean up
        deallocate(b_n,b_g)

#if ldebug
        if (debug_calc_derived_q) then
            ! plot them
            ierr = plot_derived_q(grid_eq,eq_2)
            chckerr('')

            call writo('Testing consistency of derived quantities')
            call lvl_ud(1)

            ! test consistency sigma and kappa_g
            ierr = test_sigma_with_kappa_g(grid_eq,eq_1,eq_2)
            chckerr('')

            ! test comparison with naive implementations
            ierr = test_kappa(grid_eq,eq_1,eq_2)
            chckerr('')
            select case (eq_style)
                case (1)                                                ! VMEC
                    ierr = test_sigma_vmec(grid_eq,eq_1,eq_2)
                    chckerr('')
                case (2)                                                ! HELENA
                    call test_s_hel(grid_eq,eq_2,rchi,zchi)
            end select

            call lvl_ud(-1)
            call writo('Testing done')
        end if
#endif

        call lvl_ud(-1)
    contains
        !> \private Calculate shear from direct formula
        subroutine calc_derived_s_direct(eq_2,S)
            use num_utilities, only: c
            use num_vars, only: tol_zero

            ! input / output
            type(eq_2_type), intent(in), target :: eq_2                 !< metric equilibrium variables
            real(dp), intent(out) :: s(:,:,:)                           !< shear

            ! local variables
            real(dp), pointer :: j(:,:,:) => null()                     ! jac
            real(dp), pointer :: h12(:,:,:) => null()                   ! h^alpha,psi
            real(dp), pointer :: d3h12(:,:,:) => null()                 ! D_theta h^alpha,psi
            real(dp), pointer :: h22(:,:,:) => null()                   ! h^psi,psi
            real(dp), pointer :: d3h22(:,:,:) => null()                 ! D_theta h^psi,psi
            real(dp), allocatable :: h22_corrected(:,:,:)               ! to avoid division by zero

            ! set up submatrices
            j => eq_2%jac_FD(:,:,:,0,0,0)
            h12 => eq_2%h_FD(:,:,:,c([1,2],.true.),0,0,0)
```

```
            d3h12 => eq_2%h_FD(:,:,:,c([1,2],.true.),0,0,1)
            h22 => eq_2%h_FD(:,:,:,c([2,2],.true.),0,0,0)
            d3h22 => eq_2%h_FD(:,:,:,c([2,2],.true.),0,0,1)
            allocate(h22_corrected(size(s,1),size(s,2),size(s,3)))
            h22_corrected = sign(max(abs(h22),tol_zero),h22)

            ! Calculate the shear S
            s = -(d3h12/h22_corrected - d3h22*h12/h22_corrected**2)/j

            ! clean up
            nullify(j,h12,d3h12,h22,d3h22)
        end subroutine calc_derived_s_direct

        !> \private Calculate shear by numerically derivating for HELENA.
        integer function calc_derived_s_from_deriv_hel(grid_eq,eq_2,bcs,&
            &bcs_val,S) result(ierr)

            use num_utilities, only: c, spline
            use helena_vars, only: chi_h

            character(*), parameter :: rout_name = &
                &'calc_derived_S_from_deriv_HEL'

            ! input / output
            type(grid_type), intent(in) :: grid_eq                          !< equilibrium grid
            type(eq_2_type), intent(in), target :: eq_2                     !< metric equilibrium variables
            integer :: bcs(2)                                               !< boundary conditions (theta(even),
    theta(odd))
            real(dp) :: bcs_val(2)                                          !< values for boundary conditions
            real(dp), intent(out) :: s(:,:,:)                               !< shear

            ! local variables
            integer :: jd, kd                                               ! counters
            real(dp), pointer :: j(:,:,:) => null()                         ! jac
            real(dp), pointer :: h12(:,:,:) => null()                       ! h^alpha,psi
            real(dp), pointer :: h22(:,:,:) => null()                       ! h^psi,psi

            ! initialize ierr
            ierr = 0

            ! set up submatrices
            j => eq_2%jac_FD(:,:,:,0,0,0)
            h12 => eq_2%h_FD(:,:,:,c([1,2],.true.),0,0,0)
            h22 => eq_2%h_FD(:,:,:,c([2,2],.true.),0,0,0)

            ! Calculate the shear S
            do kd = 1,grid_eq%loc_n_r
                do jd = 1,grid_eq%n(2)
                    ierr = spline(chi_h,h12(:,jd,kd)/h22(:,jd,kd),chi_h,&
                        &s(:,jd,kd),ord=3,deriv=1,bcs=bcs,bcs_val=bcs_val)
                    chckerr('')
                end do
            end do
            s = -s/j

            ! clean up
            nullify(j,h12,h22)
        end function calc_derived_s_from_deriv_hel

        !> \private Calculate shear from sigma using identity
        !! J|nabla psi|^2 S + mu_0 J B^2 sigma = K
        !! with K = -2F/R (Z(1)/R(0) + (Z(1)R(2)-R(1)Z(2))/(R(1)^2+Z(1)^2)
        subroutine calc_derived_s_from_sigma_hel(grid_eq,eq_2,Rchi,Zchi,S)
            use num_utilities, only: c
            use helena_vars, only: rbphi_h

            ! input / output
            type(grid_type), intent(in) :: grid_eq                          !< equilibrium grid
            type(eq_2_type), intent(in), target :: eq_2                     !< metric equilibrium variables
            real(dp), intent(in) :: rchi(:,:,0:)                            !< chi and chi^2 derivatives of R
            real(dp), intent(in) :: zchi(:,:,0:)                            !< chi and chi^2 derivatives of Z
            real(dp), intent(out) :: s(:,:,:)                               !< shear

            ! local variables
            integer :: jd, kd                                               ! counters
            integer :: kd_h                                                 ! kd in Helena tables
            real(dp), allocatable :: k(:,:,:)                               ! RHS
            real(dp), pointer :: j(:,:,:) => null()                         ! jac
            real(dp), pointer :: g33(:,:,:) => null()                       ! g^theta,theta
            real(dp), pointer :: h22(:,:,:) => null()                       ! h^psi,psi

            ! set up submatrices
            j => eq_2%jac_FD(:,:,:,0,0,0)
            g33 => eq_2%g_FD(:,:,:,c([3,3],.true.),0,0,0)
            h22 => eq_2%h_FD(:,:,:,c([2,2],.true.),0,0,0)

            ! set up RHS K
```

```
            allocate(k(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
            do jd = 1,grid_eq%n(2)
                k(:,jd,:) = zchi(:,:,1)/rchi(:,:,0) + &
                    &(zchi(:,:,1)*rchi(:,:,2)-rchi(:,:,1)*zchi(:,:,2))/&
                    &(rchi(:,:,1)**2+zchi(:,:,1)**2)
                do kd = 1,grid_eq%loc_n_r
                    kd_h = grid_eq%i_min-1+kd
                    k(:,jd,kd) = -2._dp*rbphi_h(kd_h,0)/rchi(:,kd,0) * &
                        &k(:,jd,kd)
                end do
            end do

            ! subtract mu_0 J B^2 sigma
            s = k - vac_perm*eq_2%sigma*g33/j

            ! divide by J |nabla psi|^2
            s = s/(j*h22)

            ! clean up
            nullify(j,g33,h22)
        end subroutine calc_derived_s_from_sigma_hel

        !> \private  Calculate Cylindrical  contravariant components  of angular
        !! derivatives of covariant parallel basis vector
        subroutine calc_derived_dc_epar(de,D_de,T_FE,D1_epar,D3_epar)
            use num_utilities, only: c

            ! input / output
            real(dp), intent(in) :: de(:,:,:,:,:,:)                             !< derivs of cov. unit vector in E (space,deriv.,unitvec,output)
            real(dp), intent(in) :: d_de(:,:,:,:,:)                             !< derivs of transf. matrix in E (space,deriv.,output)
            real(dp), intent(in) :: t_fe(:,:,:,:)                               !< transformation matrix from F to E
            real(dp), intent(out) :: d1_epar(:,:,:,:)                           !< cylindrical contravariant components of alpha derivative of parallel basis vector
            real(dp), intent(out) :: d3_epar(:,:,:,:)                           !< cylindrical contravariant components of theta derivative of parallel basis vector

            ! initialize output
            d1_epar = 0._dp
            d3_epar = 0._dp

            ! transform
            do ld = 1,3
                do id = 1,2
                    do jd = 1,2
                        ! d/dalpha
                        d1_epar(:,:,:,ld) = d1_epar(:,:,:,ld) + &
                            &de(:,:,:,id,jd,ld) * &
                            &t_fe(:,:,:,c([1,1+id],.false.)) * &                ! 1 for alpha
                            &t_fe(:,:,:,c([3,1+jd],.false.))
                        ! d/dtheta
                        d3_epar(:,:,:,ld) = d3_epar(:,:,:,ld) + &
                            &de(:,:,:,id,jd,ld) * &
                            &t_fe(:,:,:,c([3,1+id],.false.)) * &                ! 3 for theta
                            &t_fe(:,:,:,c([3,1+jd],.false.))
                    end do
                    ! d/dalpha
                    d1_epar(:,:,:,ld) = d1_epar(:,:,:,ld) + &
                        &d_de(:,:,:,id,ld) * &
                        &t_fe(:,:,:,c([1,1+id],.false.))                        ! 1 for alpha
                    ! d/dtheta
                    d3_epar(:,:,:,ld) = d3_epar(:,:,:,ld) + &
                        &d_de(:,:,:,id,ld) * &
                        &t_fe(:,:,:,c([3,1+id],.false.))                        ! 3 for theta
                end do
            end do
        end subroutine calc_derived_dc_epar

        !> \private  Calculate Equilibrium  contravariant components  of angular
        !! derivatives  of covariant  parallel basis  vector. Also  sets up  the
        !! covariant  Cylindrical components  of the  normal and  geodesic basis
        !! vectors.
        subroutine calc_derived_de_epar_vmec(grid_eq,eq_2,de,D_de,b_n,b_g)
            use num_utilities, only: c
            use num_vars, only: tol_zero

            ! input / output
            type(grid_type), intent(in) :: grid_eq                              !< equilibrium grid
            type(eq_2_type), intent(in) :: eq_2                                 !< metric equilibrium variables
            real(dp), intent(out) :: de(:,:,:,:,:,:)                            !< derivs of cov. unit vector in E (space,deriv.,unitvec,output)
            real(dp), intent(out) :: d_de(:,:,:,:,:)                            !< derivs of transf. matrix in E (space,deriv.,output)
            real(dp), intent(out) :: b_n(:,:,:,:)                               !< covariant Cylindrical components of normal basis vector
            real(dp), intent(out) :: b_g(:,:,:,:)                               !< covariant Cylindrical components of
```

```
geodesic basis vector

            ! initialize output
            de = 0._dp
            d_de = 0._dp
            b_n = 0._dp
            b_g = 0._dp

            ! calculate the derivatives of covariant unit vectors in Equilibrium
            ! coordinates

            ! d/dtheta e_theta
            de(:,:,:,1,1,1) = eq_2%R_E(:,:,:,0,2,0)                          ! ~ e_R
            !de(:,:,:,1,1,2) = 0._dp                                          ! ~ e_phi
            de(:,:,:,1,1,3) = eq_2%Z_E(:,:,:,0,2,0)                          ! ~ e_Z

            ! d/dzeta e_theta
            de(:,:,:,2,1,1) = eq_2%R_E(:,:,:,0,1,1)                          ! ~ e_R
            de(:,:,:,2,1,2) = eq_2%R_E(:,:,:,0,1,0)/eq_2%R_E(:,:,:,0,0,0)    ! ~ e_phi
            de(:,:,:,2,1,3) = eq_2%Z_E(:,:,:,0,1,1)                          ! ~ e_Z

            ! d/dtheta e_zeta
            de(:,:,:,1,2,1) = eq_2%R_E(:,:,:,0,1,1)                          ! ~ e_R
            de(:,:,:,1,2,2) = eq_2%R_E(:,:,:,0,1,0)/eq_2%R_E(:,:,:,0,0,0)    ! ~ e_phi
            de(:,:,:,1,2,3) = eq_2%Z_E(:,:,:,0,1,1)                          ! ~ e_Z

            ! d/dzeta e_zeta
            de(:,:,:,2,2,1) = eq_2%R_E(:,:,:,0,0,2)-eq_2%R_E(:,:,:,0,0,0)    ! ~ e_R
            de(:,:,:,2,2,2) = 2*eq_2%R_E(:,:,:,0,0,1)/eq_2%R_E(:,:,:,0,0,0)  ! ~ e_phi
            de(:,:,:,2,2,3) = eq_2%Z_E(:,:,:,0,0,2)                          ! ~ e_Z

            ! calculate  derivatives  of  transformation matrix  in  Equilibrium
            ! coordinates

            ! ~d/dtheta
            d_de(:,:,:,1,1) = eq_2%R_E(:,:,:,0,1,0)*&
                &eq_2%T_FE(:,:,:,c([3,2],.false.),0,1,0) + &
                &eq_2%R_E(:,:,:,0,0,1)*&
                &eq_2%T_FE(:,:,:,c([3,3],.false.),0,1,0)                     ! ~ e_R
            d_de(:,:,:,1,2) = eq_2%T_FE(:,:,:,c([3,3],.false.),0,1,0)        ! ~ e_phi
            d_de(:,:,:,1,3) = eq_2%Z_E(:,:,:,0,1,0)*&
                &eq_2%T_FE(:,:,:,c([3,2],.false.),0,1,0) + &
                &eq_2%Z_E(:,:,:,0,0,1)*&
                &eq_2%T_FE(:,:,:,c([3,3],.false.),0,1,0)                     ! ~ e_Z

            ! ~d/dzeta
            d_de(:,:,:,2,1) = eq_2%R_E(:,:,:,0,1,0)*&
                &eq_2%T_FE(:,:,:,c([3,2],.false.),0,0,1) + &
                &eq_2%R_E(:,:,:,0,0,1)*&
                &eq_2%T_FE(:,:,:,c([3,3],.false.),0,0,1)                     ! ~ e_R
            d_de(:,:,:,2,2) = eq_2%T_FE(:,:,:,c([3,3],.false.),0,0,1)        ! ~ e_phi
            d_de(:,:,:,2,3) = eq_2%Z_E(:,:,:,0,1,0)*&
                &eq_2%T_FE(:,:,:,c([3,2],.false.),0,0,1) + &
                &eq_2%Z_E(:,:,:,0,0,1)*&
                &eq_2%T_FE(:,:,:,c([3,3],.false.),0,0,1)                     ! ~ e_Z

            ! Decompose  normal and  geodesic basis  vectors into  contravariant
            ! cylindrical basis vectors.

            ! b_n
            b_n(:,:,:,2) = (eq_2%R_E(:,:,:,0,0,1)*eq_2%Z_E(:,:,:,0,1,0) - &
                &eq_2%R_E(:,:,:,0,1,0)*eq_2%Z_E(:,:,:,0,0,1))                ! store RzetaZtheta - RthetaZeta
            b_n(:,:,:,1) = &
                &-eq_2%jac_FD(:,:,:,0,0,0) * (1+eq_2%L_E(:,:,:,0,1,0)) / &
                &max(tol_zero, ( (b_n(:,:,:,2)/eq_2%R_E(:,:,:,0,0,0))**2 + &
                &eq_2%R_E(:,:,:,0,1,0)**2 + eq_2%Z_E(:,:,:,0,1,0)**2 )) / &
                &eq_2%R_E(:,:,:,0,0,0)                                       ! set up common factor of all b_n
            b_n(:,:,:,2) = b_n(:,:,:,1) * b_n(:,:,:,2)                       ! finalize b_n(2)
            b_n(:,:,:,3) = b_n(:,:,:,1) * eq_2%R_E(:,:,:,0,1,0)              ! finalize b_n(3)
            b_n(:,:,:,1) = -b_n(:,:,:,1) * eq_2%Z_E(:,:,:,0,1,0)             ! finalize b_n(1)

            ! b_g
            do kd = 1,grid_eq%loc_n_r
                b_g(:,:,kd,2) = eq_2%g_FD(:,:,kd,c([3,3],.true.),0,0,0) - &
                    &eq_2%g_FD(:,:,kd,c([1,3],.true.),0,0,0) * &
                    &eq_1%q_saf_FD(kd,0)                                     ! store J(B_theta + B_alpha q)
                b_g(:,:,kd,1) = (b_g(:,:,kd,2)*eq_2%L_E(:,:,kd,0,0,1)-&
                    &eq_2%g_FD(:,:,kd,c([1,3],.true.),0,0,0)) / &
                    &(1+eq_2%L_E(:,:,kd,0,1,0))                              ! store J((B_theta + B_alpha q) L_zeta -
B_alpha)/(1+L_theta)
            end do
            b_g(:,:,:,3) = b_g(:,:,:,1)*eq_2%Z_E(:,:,:,0,1,0) - &
                &b_g(:,:,:,2)*eq_2%Z_E(:,:,:,0,0,1)                          ! finalize b_g(3) (apart from pre-factor)
            b_g(:,:,:,1) = b_g(:,:,:,1)*eq_2%R_E(:,:,:,0,1,0) - &
                &b_g(:,:,:,2)*eq_2%R_E(:,:,:,0,0,1)                          ! finalize b_g(1) (apart from pre-factor)
            b_g(:,:,:,2) = -b_g(:,:,:,2)*eq_2%R_E(:,:,:,0,0,0)**2            ! finalize b_g(2) (apart from pre-factor)
            do ld = 1,3
```

```
                b_g(:,:,:,ld) = b_g(:,:,:,ld) / &
                    &eq_2%g_FD(:,:,:,c([3,3],.true.),0,0,0)                    ! prefactor 1/g_thetatheta
            end do
        end subroutine calc_derived_de_epar_vmec
        subroutine calc_derived_de_epar_hel(grid_eq,eq_1,Rchi,Zchi,&
            &de,D_de,b_n,b_g)

            use helena_vars, only: rbphi_h

            ! input / output
            type(grid_type), intent(in) :: grid_eq                            !< equilibrium grid
            type(eq_1_type), intent(in) :: eq_1                               !< flux equilibrium variables
            real(dp), intent(in) :: rchi(:,:,0:)                              !< chi and chi^2 derivatives of R
            real(dp), intent(in) :: zchi(:,:,0:)                              !< chi and chi^2 derivatives of Z
            real(dp), intent(out) :: de(:,:,:,:,:,:)                          !< derivs of cov. unit vector in E (space,deriv.,unitvec,output)
            real(dp), intent(out) :: d_de(:,:,:,:,:)                          !< derivs of transf. matrix in E (space,deriv.,output)
            real(dp), intent(out) :: b_n(:,:,:,:)                             !< covariant Cylindrical components of normal basis vector
            real(dp), intent(out) :: b_g(:,:,:,:)                             !< covariant Cylindrical components of geodesic basis vector

            ! local variables
            integer :: kd_h                                                   ! kd in Helena tables
            real(dp), allocatable :: dum1(:,:)                                ! dummy variable
            real(dp), allocatable :: dum2(:,:)                                ! dummy variable

            ! initialize output
            de = 0._dp
            d_de = 0._dp
            b_n = 0._dp
            b_g = 0._dp

            ! calculate the derivatives of covariant unit vectors in Equilibrium
            ! coordinates

            ! d/dtheta e_theta
            de(:,1,:,1,1,1) = rchi(:,:,2)                                     ! ~ e_R
            !de(:,1,:,1,1,2) = 0._dp                                           ! ~ e_phi
            de(:,1,:,1,1,3) = zchi(:,:,2)                                     ! ~ e_Z

            ! d/dzeta e_theta
            !de(:,1,:,2,1,1) = 0._dp                                           ! ~ e_R
            de(:,1,:,2,1,2) = -rchi(:,:,1)/rchi(:,:,0)                        ! ~ e_phi
            !de(:,1,:,2,1,3) = 0._dp                                           ! ~ e_Z

            ! d/dtheta e_zeta
            !de(:,1,:,1,2,1) = 0._dp                                           ! ~ e_R
            de(:,1,:,1,2,2) = -rchi(:,:,1)/rchi(:,:,0)                        ! ~ e_phi
            !de(:,1,:,1,2,3) = 0._dp                                           ! ~ e_Z

            ! d/dzeta e_zeta
            de(:,1,:,2,2,1) = -rchi(:,:,0)                 ! ~ e_R
            !de(:,1,:,2,2,2) = 0._dp                                           ! ~ e_phi
            !de(:,1,:,2,2,3) = 0._dp                                           ! ~ e_Z

            ! Decompose  normal and  geodesic basis  vectors into  contravariant
            ! cylindrical basis vectors.
            do jd = 1,grid_eq%n(2)
                ! auxiliary variables
                allocate(dum1(grid_eq%n(1),grid_eq%loc_n_r))                  ! Rchi^2 + Zchi^2
                allocate(dum2(grid_eq%n(1),grid_eq%loc_n_r))                  ! Rchi^2 + Zchi^2 + (qR)^2
                dum1 = rchi(:,:,1)**2 + zchi(:,:,1)**2
                do kd = 1,grid_eq%loc_n_r
                    dum2(:,kd) = dum1(:,kd) + &
                        &(eq_1%q_saf_FD(kd,0)*rchi(:,kd,0))**2
                end do

                ! b_n
                b_n(:,jd,:,1) = rchi(:,:,0)*zchi(:,:,1)/dum1
                b_n(:,jd,:,3) = -rchi(:,:,0)*rchi(:,:,1)/dum1

                ! b_g
                b_g(:,jd,:,1) = -rchi(:,:,0)**2 * rchi(:,:,1)/dum2
                b_g(:,jd,:,2) = -rchi(:,:,0)**2 * dum1/dum2
                b_g(:,jd,:,3) = -rchi(:,:,0)**2 * zchi(:,:,1)/dum2

                ! clean up
                deallocate(dum1,dum2)
            end do
            do kd = 1,grid_eq%loc_n_r
                kd_h = grid_eq%i_min-1+kd
                b_n(:,:,kd,:) = b_n(:,:,kd,:) * &
                    &eq_1%q_saf_FD(kd,0)/rbphi_h(kd_h,0)
                b_g(:,:,kd,1:3:2) = b_g(:,:,kd,1:3:2) * eq_1%q_saf_FD(kd,0)     ! not b_g(2)
            end do
```

```fortran
        end subroutine calc_derived_de_epar_hel

        !> \private  Calculate normal  and geodesic  curvature components  using
        !! VMEC method.
        subroutine calc_derived_kappa_from_e(grid_eq,eq_1,eq_2,b_n,b_g,&
            &kappa_n,kappa_g)

            use num_utilities, only: c

            ! input / output
            type(grid_type), intent(in) :: grid_eq                          !< equilibrium grid
            type(eq_1_type), intent(in) :: eq_1                             !< flux equilibrium variables
            type(eq_2_type), intent(in) :: eq_2                             !< metric equilibrium variables
            real(dp), intent(in) :: b_n(:,:,:,:)                            !< covariant Cylindrical components of normal basis vector
            real(dp), intent(in) :: b_g(:,:,:,:)                            !< covariant Cylindrical components of geodesic basis vector
            real(dp), intent(out) :: kappa_n(:,:,:)                         !< normal curvature
            real(dp), intent(out) :: kappa_g(:,:,:)                         !< geodesic curvature

            ! calculate curvature: dot d/dtheta e_theta and basis vectors
            kappa_n = 0._dp
            kappa_g = 0._dp
            do ld = 1,3
                kappa_n = kappa_n + d3_epar(:,:,:,ld) * b_n(:,:,:,ld)
                kappa_g = kappa_g + d3_epar(:,:,:,ld) * b_g(:,:,:,ld)
            end do

            ! divide by |e_theta|^2
            kappa_n = kappa_n / eq_2%g_FD(:,:,:,c([3,3],.true.),0,0,0)
            kappa_g = kappa_g / eq_2%g_FD(:,:,:,c([3,3],.true.),0,0,0)

            ! possibly correct for toroidal flux
            if (.not.use_pol_flux_f) then
                do kd = 1,grid_eq%loc_n_r
                    kappa_n(:,:,kd) = kappa_n(:,:,kd) * eq_1%q_saf_E(kd,0)
                    kappa_g(:,:,kd) = kappa_g(:,:,kd) / eq_1%q_saf_E(kd,0)
                end do
            end if
        end subroutine calc_derived_kappa_from_e

        !> \private  Calculate parallel current using VMEC method.
        subroutine calc_derived_sigma_from_e(eq_2,b_n,sigma)
            use num_utilities, only: c

            ! input / output
            type(eq_2_type), intent(in) :: eq_2                             !< metric equilibrium variables
            real(dp), intent(in) :: b_n(:,:,:,:)                            !< covariant Cylindrical components of normal basis vector
            real(dp), intent(out) :: sigma(:,:,:)                           !< parallel current

            ! calculate the parallel current
            sigma = 0._dp
            do ld = 1,3
                sigma = sigma + 2._dp * b_n(:,:,:,ld) * &
                    &(d3_epar(:,:,:,ld) * &
                    &eq_2%g_FD(:,:,:,c([1,3],.true.),0,0,0) - &
                    &d1_epar(:,:,:,ld) * &
                    &eq_2%g_FD(:,:,:,c([3,3],.true.),0,0,0)) / &
                    &eq_2%jac_FD(:,:,:,0,0,0)**2
            end do

            ! add shear term and divide by -B_theta mu_0
            sigma = -(sigma + eq_2%jac_FD(:,:,:,0,0,0) * &
                &eq_2%h_FD(:,:,:,c([2,2],.true.),0,0,0)*eq_2%S) * &
                &eq_2%jac_FD(:,:,:,0,0,0)/&
                &(vac_perm*eq_2%g_FD(:,:,:,c([3,3],.true.),0,0,0))
        end subroutine calc_derived_sigma_from_e

        !> \private  Calculate parallel current using HELENA method.
        subroutine calc_derived_sigma_hel(grid_eq,eq_2,sigma)
            use helena_vars, only: rbphi_h
            use num_utilities, only: c

            ! input / output
            type(grid_type), intent(in) :: grid_eq                          !< equilibrium grid
            type(eq_2_type), intent(in), target :: eq_2                     !< metric equilibrium variables
            real(dp), intent(out) :: sigma(:,:,:)                           !< parallel current

            ! local variables
            integer :: kd                                                   ! counter
            integer :: kd_h                                                 ! kd in Helena tables
            real(dp), pointer :: j(:,:,:) => null()                         ! jac
            real(dp), pointer :: g13(:,:,:) => null()                       ! g_alpha,theta
            real(dp), pointer :: g33(:,:,:) => null()                       ! g_theta,theta

            ! set up submatrices
```

```
            j => eq_2%jac_FD(:,:,:,0,0,0)
            g13 => eq_2%g_FD(:,:,:,c([1,3],.true.),0,0,0)
            g33 => eq_2%g_FD(:,:,:,c([3,3],.true.),0,0,0)

            ! calculate parallel current using direct formula
            do kd = 1,grid_eq%loc_n_r
                kd_h = grid_eq%i_min-1+kd
                sigma(:,:,kd) = -rbphi_h(kd_h,1) / vac_perm - &
                    &eq_1%pres_FD(kd,1)*j(:,:,kd)*g13(:,:,kd)/g33(:,:,kd)
            end do

            ! clean up
            nullify(j,g13,g33)
        end subroutine calc_derived_sigma_hel

#if ldebug
        !> \private Plot derived equilibrium quantities for debug.
        integer function plot_derived_q(grid_eq,eq_2) result(ierr)
            use num_vars, only: prog_style, eq_jobs_lims, eq_job_nr, &
                &use_normalization, mu_0_original
            use grid_utilities, only: trim_grid, calc_xyz_grid
            use grid_vars, only: alpha
            use eq_vars, only: max_flux_f, b_0, r_0

            character(*), parameter :: rout_name = 'plot_derived_q'

            ! input / output
            type(grid_type), intent(in) :: grid_eq                              !< equilibrium grid
            type(eq_2_type), intent(in) :: eq_2                                 !< metric equilibrium variables

            ! local variables
            type(grid_type) :: grid_trim                                        ! trimmed equilibrium grid
            real(dp) :: norm_factors(4)                                         ! normalization factors
            real(dp), allocatable :: x_plot(:,:,:)                              ! x values of total plot
            real(dp), allocatable :: y_plot(:,:,:)                              ! y values of total plot
            real(dp), allocatable :: z_plot(:,:,:)                              ! z values of total plot
            real(dp), pointer :: ang_par_f(:,:,:) => null()                     ! parallel angle theta_F or zeta_F
            integer :: norm_id(2)                                               ! untrimmed normal indices for trimmed grids
            integer :: plot_dim(3)                                              ! dimensions of plot
            integer :: plot_offset(3)                                           ! local offset of plot
            logical :: cont_plot                                                ! continued plot

            ! initialize ierr
            ierr = 0

            call writo('Plotting derived equilibrium quantities')
            call lvl_ud(1)

            ! trim equilibrium grid
            ierr = trim_grid(grid_eq,grid_trim,norm_id)
            chckerr('')

            ! allocate variables
            allocate(x_plot(grid_trim%n(1),grid_trim%n(2),grid_trim%loc_n_r))
            allocate(y_plot(grid_trim%n(1),grid_trim%n(2),grid_trim%loc_n_r))
            allocate(z_plot(grid_trim%n(1),grid_trim%n(2),grid_trim%loc_n_r))

            ! point parallel angle
            if (use_pol_flux_f) then
                ang_par_f => grid_eq%theta_F
            else
                ang_par_f => grid_eq%zeta_F
            end if

            ! calculate grid
            select case (prog_style)
                case (1)                                                        ! PB3D
                    select case (eq_style)
                        case (1)                                                ! VMEC
                            x_plot = ang_par_f(:,:,norm_id(1):norm_id(2))
                            do jd = 1,grid_trim%n(2)
                                y_plot(:,jd,:) = alpha(jd)
                            end do
                            do kd = 1,grid_trim%loc_n_r
                                z_plot(:,:,kd) = &
                                    &grid_trim%r_F(kd)*2*pi/max_flux_f
                            end do
                        case (2)                                                ! HELENA
                            ierr = calc_xyz_grid(grid_eq,grid_trim,x_plot,&
                                &y_plot,z_plot)
                            chckerr('')
                    end select
                case (2)                                                        ! POST
                    ierr = calc_xyz_grid(grid_eq,grid_trim,x_plot,y_plot,z_plot)
                    chckerr('')
            end select

```

```fortran
        ! set up plot dimensions and offset
        select case (eq_style)
            case (1)                                                    ! VMEC
                plot_dim = [eq_jobs_lims(2,size(eq_jobs_lims,2)) - &
                    &eq_jobs_lims(1,1) + 1, grid_trim%n(2:3)]
                plot_offset = [eq_jobs_lims(1,eq_job_nr)-1,0,&
                    &grid_trim%i_min-1]
                cont_plot = eq_job_nr.gt.1
            case (2)                                                    ! HELENA
                plot_dim = grid_trim%n
                plot_offset = [0,0,grid_trim%i_min-1]
                cont_plot = .false.
        end select

        ! set up normalization factors
        norm_factors = 1._dp
        if (use_normalization) then
            norm_factors(1) = 1._dp/(mu_0_original*r_0)                 ! sigma
            norm_factors(2) = 1._dp/(r_0**3)                            ! S
            norm_factors(3) = 1._dp/(r_0*b_0)                           ! kappa_n
            norm_factors(4) = 1._dp                                     ! kappa_g
        end if

        ! plot sigma
        call plot_hdf5('sigma','TEST_sigma',&
            &eq_2%sigma(:,:,norm_id(1):norm_id(2))*norm_factors(1),&
            &tot_dim=plot_dim,loc_offset=plot_offset,cont_plot=cont_plot,&
            &x=x_plot,y=y_plot,z=z_plot)

        ! plot shear
        call plot_hdf5('shear','TEST_S',&
            &eq_2%S(:,:,norm_id(1):norm_id(2))*norm_factors(2),&
            &tot_dim=plot_dim,loc_offset=plot_offset,cont_plot=cont_plot,&
            &x=x_plot,y=y_plot,z=z_plot)

        ! plot kappa_n
        call plot_hdf5('kappa_n','TEST_kappa_n',&
            &eq_2%kappa_n(:,:,norm_id(1):norm_id(2))*norm_factors(3),&
            &tot_dim=plot_dim,loc_offset=plot_offset,cont_plot=cont_plot,&
            &x=x_plot,y=y_plot,z=z_plot)

        ! plot kappa_g
        call plot_hdf5('kappa_g','TEST_kappa_g',&
            &eq_2%kappa_g(:,:,norm_id(1):norm_id(2))*norm_factors(4),&
            &tot_dim=plot_dim,loc_offset=plot_offset,cont_plot=cont_plot,&
            &x=x_plot,y=y_plot,z=z_plot)

        ! clean up
        nullify(ang_par_f)
        call grid_trim%dealloc()

        call lvl_ud(-1)
    end function plot_derived_q

    !> \private test whether -2 p' J kappa_g = D3sigma
    integer function test_sigma_with_kappa_g(grid_eq,eq_1,eq_2) result(ierr)
        use num_utilities, only: spline, calc_int
        use num_vars, only: norm_disc_prec_eq

        character(*), parameter :: rout_name = 'test_sigma_with_kappa_g'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                          !< equilibrium grid
        type(eq_1_type), intent(in) :: eq_1                             !< flux equilibrium variables
        type(eq_2_type), intent(in) :: eq_2                             !< metric equilibrium variables

        ! local variables
        real(dp), allocatable :: sigma_alt(:,:,:)                       ! alternative sigma
        real(dp), allocatable :: d3sigma(:,:,:)                         ! D_theta sigma
        real(dp), allocatable :: d3sigma_alt(:,:,:)                     ! alternative D_theta sigma
        real(dp), pointer :: ang_par_f(:,:,:) => null()                 ! parallel angle theta_F or zeta_F

        ! initialize ierr
        ierr = 0

        call writo('Testing whether -2 p'' J kappa_g = D3sigma')
        call lvl_ud(1)

        ! point parallel angle
        if (use_pol_flux_f) then
            ang_par_f => grid_eq%theta_F
        else
            ang_par_f => grid_eq%zeta_F
        end if

        ! allocate variables
        allocate(d3sigma(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
```

```
            allocate(d3sigma_alt(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
            allocate(sigma_alt(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))

            ! get derivative of sigma
            do kd = 1,grid_eq%loc_n_r
                do jd = 1,grid_eq%n(2)
                    ierr = spline(ang_par_f(:,jd,kd),eq_2%sigma(:,jd,kd),&
                        &ang_par_f(:,jd,kd),d3sigma(:,jd,kd),&
                        &ord=norm_disc_prec_eq,deriv=1)
                    chckerr('')
                end do
            end do

            ! calculate alternatively derived sigma
            do kd = 1,grid_eq%loc_n_r
                d3sigma_alt(:,:,kd) = -2*eq_1%pres_FD(kd,1)*&
                    &eq_2%kappa_g(:,:,kd)*eq_2%jac_FD(:,:,kd,0,0,0)
            end do

            ! calculate alternative sigma by integration
            ! Note:  there  is  an  undetermined constant  of  integration  that
            ! depends on the normal coordinate and the geodesic coordinate.
            do kd = 1,grid_eq%loc_n_r
                do jd = 1,grid_eq%n(2)
                    ierr = calc_int(d3sigma_alt(:,jd,kd),ang_par_f(:,jd,kd),&
                        &sigma_alt(:,jd,kd))
                    chckerr('')
                end do
                sigma_alt(:,:,kd) = eq_2%sigma(1,1,kd) + sigma_alt(:,:,kd)
            end do

            ! plot output
            call plot_diff_hdf5(d3sigma,d3sigma_alt,&
                &'TEST_diff_D3sigma_through_kappa_g',&
                &grid_eq%n,[0,0,grid_eq%i_min-1],&
                &descr='To test whether -2 p'' J kappa_g = D3sigma',&
                &output_message=.true.)
            call plot_diff_hdf5(eq_2%sigma,sigma_alt,&
                &'TEST_diff_sigma_through_kappa_g',&
                &grid_eq%n,[0,0,grid_eq%i_min-1],descr='To test whether &
                &int(-2 p'' J kappa_g) = sigma',output_message=.true.)

            ! clean up
            nullify(ang_par_f)

            call lvl_ud(-1)
        end function test_sigma_with_kappa_g

        !> \private test agreement between curvatures and naive implementation
        integer function test_kappa(grid_eq,eq_1,eq_2) result(ierr)
            use num_utilities, only: c

            character(*), parameter :: rout_name = 'test_kappa'

            ! input / output
            type(grid_type), intent(in) :: grid_eq                          !< equilibrium grid
            type(eq_1_type), intent(in) :: eq_1                             !< flux equilibrium variables
            type(eq_2_type), intent(in), target :: eq_2                     !< metric equilibrium variables

            ! local variables
            real(dp), allocatable :: kappa_alt(:,:,:,:)                     ! alternative kappa_n (1) and kappa_g(2)
            real(dp), pointer :: j(:,:,:) => null()                         ! jac
            real(dp), pointer :: d1j(:,:,:) => null()                       ! D_alpha jac
            real(dp), pointer :: d2j(:,:,:) => null()                       ! D_psi jac
            real(dp), pointer :: d3j(:,:,:) => null()                       ! D_theta jac
            real(dp), pointer :: g13(:,:,:) => null()                       ! g_alpha,theta
            real(dp), pointer :: g33(:,:,:) => null()                       ! g_theta,theta
            real(dp), pointer :: d1g33(:,:,:) => null()                     ! D_alpha g_theta,theta
            real(dp), pointer :: d2g33(:,:,:) => null()                     ! D_psi g_theta,theta
            real(dp), pointer :: d3g33(:,:,:) => null()                     ! D_theta g_theta,theta
            real(dp), pointer :: h12(:,:,:) => null()                       ! h_alpha,psi
            real(dp), pointer :: h22(:,:,:) => null()                       ! h_psi,psi
            real(dp), pointer :: h23(:,:,:) => null()                       ! h_psi,theta

            ! initialize ierr
            ierr = 0

            call writo('Testing whether kappa agrees with naive implementation')
            call lvl_ud(1)

            ! set up submatrices
            j => eq_2%jac_FD(:,:,:,0,0,0)
            d1j => eq_2%jac_FD(:,:,:,1,0,0)
            d2j => eq_2%jac_FD(:,:,:,0,1,0)
            d3j => eq_2%jac_FD(:,:,:,0,0,1)
            g13 => eq_2%g_FD(:,:,:,c([1,3],.true.),0,0,0)
            g33 => eq_2%g_FD(:,:,:,c([3,3],.true.),0,0,0)
```

```
            d1g33 => eq_2%g_FD(:,:,:,c([3,3],.true.),1,0,0)
            d2g33 => eq_2%g_FD(:,:,:,c([3,3],.true.),0,1,0)
            d3g33 => eq_2%g_FD(:,:,:,c([3,3],.true.),0,0,1)
            h12 => eq_2%h_FD(:,:,:,c([1,2],.true.),0,0,0)
            h22 => eq_2%h_FD(:,:,:,c([2,2],.true.),0,0,0)
            h23 => eq_2%h_FD(:,:,:,c([2,3],.true.),0,0,0)

            ! initialize
            allocate(kappa_alt(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r,2))
            kappa_alt = 0._dp

            ! Calculate naive normal curvature kappa_n
            do kd = 1,grid_eq%loc_n_r
                kappa_alt(:,:,kd,1) = &
                    &vac_perm*j(:,:,kd)**2*eq_1%pres_FD(kd,1)/g33(:,:,kd) + &
                    &1._dp/(2*h22(:,:,kd)) * ( &
                    &h12(:,:,kd) * ( d1g33(:,:,kd)/g33(:,:,kd) - &
                    &2*d1j(:,:,kd)/j(:,:,kd) ) + &
                    &h22(:,:,kd) * ( d2g33(:,:,kd)/g33(:,:,kd) - &
                    &2*d2j(:,:,kd)/j(:,:,kd) ) + &
                    &h23(:,:,kd) * ( d3g33(:,:,kd)/g33(:,:,kd) - &
                    &2*d3j(:,:,kd)/j(:,:,kd) ) )
            end do

            ! Calculate naive geodesic curvature kappa_g
            kappa_alt(:,:,:,2) = (0.5*d1g33/g33 - d1j/j) - &
                &g13/g33*(0.5*d3g33/g33 - d3j/j)

            call plot_diff_hdf5(eq_2%kappa_n,kappa_alt(:,:,:,1),&
                &'TEST_diff_kappa_n',grid_eq%n,[0,0,grid_eq%i_min-1],&
                &descr='To test whether kappa_n agrees with naive calculation',&
                &output_message=.true.)
            call plot_diff_hdf5(eq_2%kappa_g,kappa_alt(:,:,:,2),&
                &'TEST_diff_kappa_g',grid_eq%n,[0,0,grid_eq%i_min-1],&
                &descr='To test whether kappa_g agrees with naive calculation',&
                &output_message=.true.)

            !!! temporary: for 2018 paper
            !!ierr = plot_diff_for_paper(grid_eq%loc_r_F,eq_2%kappa_n,&
                !!&kappa_ALT(:,:,:,1),'kappa_n')
            !!CHCKERR('')
            !!ierr = plot_diff_for_paper(grid_eq%loc_r_F,eq_2%kappa_g,&
                !!&kappa_ALT(:,:,:,2),'kappa_g')
            !!CHCKERR('')

            ! clean up
            nullify(j,d1j,d2j,d3j,g13,g33,d1g33,d2g33,d3g33,h12,h22,h23)

            call lvl_ud(-1)
        end function test_kappa

        !> \private  test  agreement  between  parallel  current  and  naive
        !! implementation for VMEC
        integer function test_sigma_vmec(grid_eq,eq_1,eq_2) result(ierr)
            use num_utilities, only: spline, calc_int, c
            use num_vars, only: norm_disc_prec_eq
            use vmec_vars, only: b_v_sub_s, b_v_sub_c, is_asym_v
            use vmec_utilities, only: fourier2real

            character(*), parameter :: rout_name = 'test_sigma_VMEC'

            ! input / output
            type(grid_type), intent(in) :: grid_eq                          !< equilibrium grid
            type(eq_1_type), intent(in) :: eq_1                             !< flux equilibrium variables
            type(eq_2_type), intent(in), target :: eq_2                     !< metric equilibrium variables

            ! local variables
            real(dp), allocatable :: sigma_alt(:,:,:)                       ! alternative sigma
            real(dp), allocatable :: b_v(:,:,:,:)                           ! magnetic field in VMEC coordinates
            real(dp), allocatable :: b_alpha(:,:,:)                         ! B_alpha Flux coordinates
            real(dp), pointer :: j(:,:,:) => null()                         ! jac
            real(dp), pointer :: d1j(:,:,:) => null()                       ! D_alpha jac
            real(dp), pointer :: g13(:,:,:) => null()                       ! g_alpha,theta
            real(dp), pointer :: g23(:,:,:) => null()                       ! g_psi,theta
            real(dp), pointer :: d1g23(:,:,:) => null()                     ! D_alpha g_psi,theta
            real(dp), pointer :: g33(:,:,:) => null()                       ! g_theta,theta

            ! initialize ierr
            ierr = 0

            call writo('Testing whether sigma agrees with naive implementation')
            call lvl_ud(1)

            ! set up submatrices
            j => eq_2%jac_FD(:,:,:,0,0,0)
            d1j => eq_2%jac_FD(:,:,:,1,0,0)
            g13 => eq_2%g_FD(:,:,:,c([1,3],.true.),0,0,0)
```

```
            g23 => eq_2%g_FD(:,:,:,c([2,3],.true.),0,0,0)
            d1g23 => eq_2%g_FD(:,:,:,c([2,3],.true.),1,0,0)
            g33 => eq_2%g_FD(:,:,:,c([3,3],.true.),0,0,0)

            ! initialize
            allocate(sigma_alt(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
            sigma_alt = 0._dp

            ! magnetic field in VMEC coordinates
            allocate(b_v(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r,2:3))
            do id = 2,3                                                     ! only angular V components count for e_alpha
                ierr = fourier2real(&
                    &b_v_sub_c(:,grid_eq%i_min:grid_eq%i_max,id),&
                    &b_v_sub_s(:,grid_eq%i_min:grid_eq%i_max,id),&
                    &grid_eq%trigon_factors,b_v(:,:,:,id),&
                    &[.true.,is_asym_v])
                chckerr('')
            end do

            ! transform them to Flux coord. system
            allocate(b_alpha(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
            b_alpha = 0._dp
            do kd = 2,3
                b_alpha = b_alpha + b_v(:,:,:,kd) * &
                    &eq_2%T_FE(:,:,:,c([1,kd],.false.),0,0,0)
            end do

            !!! more elegant but less accurate alternative:
            !!B_alpha = g13/J

            ! derivate in normal direction
            do jd = 1,grid_eq%n(2)
                do id = 1,grid_eq%n(1)
                    ierr = spline(grid_eq%loc_r_F,b_alpha(id,jd,:),&
                        &grid_eq%loc_r_F,sigma_alt(id,jd,:),&
                        &ord=norm_disc_prec_eq,deriv=1)
                    chckerr('')
                end do
            end do

            ! contribute to sigma
            sigma_alt = (d1g23 - g23*d1j/j)/j - sigma_alt

            !!! equally elegant but also inaccurate second alternative:
            !!sigma_ALT = (D1g23 - g23*D1J/J)/J - (D2g13 - g13*D2J/J)/J
            do kd = 1,grid_eq%loc_n_r
                sigma_alt(:,:,kd) = sigma_alt(:,:,kd) / vac_perm - &
                    &eq_1%pres_FD(kd,1)*j(:,:,kd)*g13(:,:,kd)/g33(:,:,kd)
            end do

            call plot_diff_hdf5(eq_2%sigma,sigma_alt,'TEST_diff_sigma',&
                &grid_eq%n,[0,0,grid_eq%i_min-1],descr='To test whether &
                &sigma agrees with naive calculation',output_message=.true.)

            !!! temporary: for 2018 paper
            !!ierr = plot_diff_for_paper(grid_eq%loc_r_F,eq_2%sigma,sigma_ALT,&
                !!&'sigma')
            !!CHCKERR('')

            ! clean up
            nullify(j,d1j,g13,g23,d1g23,g33)

            call lvl_ud(-1)
        end function test_sigma_vmec

        !> \private test agreement between shear and implementation using
        !! identity to relate to sigma
        subroutine test_s_hel(grid_eq,eq_2,Rchi,Zchi)
            ! input / output
            type(grid_type), intent(in) :: grid_eq                          !< equilibrium grid
            type(eq_2_type), intent(in) :: eq_2                             !< metric equilibrium variables
            real(dp), intent(in) :: rchi(:,:,0:)                            !< chi and chi^2 derivatives of R
            real(dp), intent(in) :: zchi(:,:,0:)                            !< chi and chi^2 derivatives of Z

            ! local variables
            real(dp), allocatable :: s_alt(:,:,:)                           ! alternative shear

            call writo('Testing whether shear agrees with alternative &
                &implementation using sigma')
            call lvl_ud(1)

            allocate(s_alt(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))

            ! calculate implementation with identity using sigma
            call calc_derived_s_from_sigma_hel(grid_eq,eq_2,rchi,zchi,s_alt)
            call plot_diff_hdf5(eq_2%S,s_alt,'TEST_diff_S_from_sigma',&
```

```
                &grid_eq%n,[0,0,grid_eq%i_min-1],descr='To test whether &
                &sigma agrees with naive calculation',output_message=.true.)

            ! calculate naive implementation
            call calc_derived_s_direct(eq_2,s_alt)
            call plot_diff_hdf5(eq_2%S,s_alt,'TEST_diff_S_naive',&
                &grid_eq%n,[0,0,grid_eq%i_min-1],descr='To test whether &
                &sigma agrees with naive calculation',output_message=.true.)

            call lvl_ud(-1)
        end subroutine test_s_hel

        !> \private make plot for 2018 paper; requires one process only.
        integer function plot_diff_for_paper(r, A, B, title) result(ierr)
            use num_vars, only: n_procs

            character(*), parameter :: rout_name = 'plot_diff_for_paper'

            ! input / output
            real(dp), intent(in) :: r(:)                                    !< r at which tabulated
            real(dp), intent(in) :: a(:,:,:)                                !< GOOD value (optimized implementation)
            real(dp), intent(in) :: b(:,:,:)                                !< BAD value (naive implementation)
            character(len=*), intent(in) :: title                           !< plot title

            ! local variables
            integer :: jd, kd                                               ! counters
            integer :: n_r                                                  ! size of r
            integer :: n_par                                                ! number of parallel points
            character(len=max_str_ln) :: plot_title(3)                      ! titles of plot (GOOD, BAD, diff)

            ! initialize ierr
            ierr = 0

            if (n_procs.gt.1) then
                ierr = 1
                chckerr('Need 1 process')
            end if

            n_r = size(r)
            n_par = size(a, 1)

            do jd = 1,size(a,2)
                plot_title(1) = trim(title)//'_GOOD'
                plot_title(2) = trim(title)//'_BAD'
                plot_title(3) = trim(title)//'_DIFF'
                if (jd.gt.1) then
                    do kd = 1,3
                        plot_title(kd) = trim(plot_title(kd))//'_'//&
                            &trim(i2str(kd))
                    end do
                end if
                call print_ex_2d(['''],plot_title(1),transpose(a(:,jd,:)),&
                    &x=reshape(r*2*pi/max_flux_f,[n_r,1]),draw=.false.)
                call draw_ex(['''],plot_title(1),n_par,1,.false.)
                call print_ex_2d(['''],plot_title(2),transpose(b(:,jd,:)),&
                    &x=reshape(r*2*pi/max_flux_f,[n_r,1]),draw=.false.)
                call draw_ex(['''],plot_title(2),n_par,1,.false.)
                call print_ex_2d(['''],plot_title(3),&
                    &transpose(abs(b(:,jd,:)-a(:,jd,:))/&
                    &(abs(b(:,jd,:))+abs(a(:,jd,:)))),&
                    &x=reshape(r*2*pi/max_flux_f,[n_r,1]),draw=.false.)
                call draw_ex(['''],plot_title(3),n_par,1,.false.)
            end do
        end function plot_diff_for_paper
#endif
    end function calc_derived_q

    !> Sets up normalization constants.
    !!
    !!  - VMEC version (\c eq_style=1):\n
    !!      Normalization depends on \c norm_style:
    !!      -# MISHKA
    !!          - R_0:    major radius (= average magnetic axis)
    !!          - B_0:    B on magnetic axis (theta = zeta = 0)
    !!          - pres_0: reference pressure (= B_0^2/mu_0)
    !!          - psi_0:  reference flux (= R_0^2 B_0)
    !!          - rho_0:  reference mass density
    !!      -# COBRA
    !!          - R_0:    major radius (= average geometric axis)
    !!          - pres_0: pressure on magnetic axis
    !!          - B_0:    reference magnetic field (= sqrt(2pres_0mu_0 / beta))
    !!          - psi_0:  reference flux (= R_0^2 B_0 / aspr^2)
    !!          - rho_0:  reference mass density
    !!      where aspr (aspect ratio) and beta are given by VMEC.
    !!  - HELENA version (\c eq_style=2):\n
    !!      MISHKA Normalization  is  used  by  default  and  does  not  depend
    !!      on  \c norm_style
```

```
    !!
    !! \see read_hel()
    !!
    !! \note \c rho_0  is not given through by the  equilibrium codes and should
    !! be user-supplied
    !!
    !! \return ierr
    integer function calc_normalization_const() result(ierr)
        use num_vars, only: rank, eq_style, mu_0_original, use_normalization, &
            &rich_restart_lvl
        use eq_vars, only: t_0, b_0, pres_0, psi_0, r_0, rho_0

        character(*), parameter :: rout_name = 'calc_normalization_const'

        ! local variables
        integer :: nr_overridden_const                                  ! nr. of user-overridden constants, to print warning if > 0
        character(len=max_str_ln) :: err_msg                            ! error message

        ! initialize ierr
        ierr = 0

        ! initialize BR_normalization_provided
        br_normalization_provided = [.false.,.false.]

        if (use_normalization) then
            ! user output
            call writo('Calculating the normalization constants')
            call lvl_ud(1)

            ! initialize nr_overridden_const
            nr_overridden_const = 0

            ! calculation
            if (rank.eq.0) then
                ! choose which equilibrium style is being used:
                !   1:  VMEC
                !   2:  HELENA
                select case (eq_style)
                    case (1)                                            ! VMEC
                        call calc_normalization_const_vmec
                    case (2)                                            ! HELENA
                        call calc_normalization_const_hel
                end select
            end if

            ! print warning if user-overridden
            if (nr_overridden_const.gt.0) &
                &call writo(trim(i2str(nr_overridden_const))//&
                &' constants were overridden by user. Consistency is NOT &
                &checked!',warning=.true.)

            ! print constants
            call print_normalization_const(r_0,rho_0,b_0,pres_0,psi_0,&
                &mu_0_original,t_0)

            ! check whether it is physically consistent
            if (t_0.lt.0._dp) then
                ierr = 1
                err_msg = 'Alfven time is negative. Are you sure the &
                    &equilibrium is consistent?'
                chckerr(err_msg)
            end if

            ! user output
            call lvl_ud(-1)
            call writo('Normalization constants calculated')
        else if (rich_restart_lvl.eq.1) then                            ! only for first Richardson leevel
            ! user output
            call writo('Normalization not used')
        end if
    contains
        ! VMEC version
        !> \private
        subroutine calc_normalization_const_vmec
            use num_vars, only: norm_style
            use vmec_vars, only: r_v_c, b_0_v, rmax_surf, rmin_surf, pres_v, &
                &beta_v
            !use VMEC_vars, only: aspr_V

            select case (norm_style)
                ! Note that PB3D does not run with the exact COBRA normalization
                ! (for style 2), as it is not a pure nondimensionalization, that
                ! would  modify the  equations by  intrucing extra  factors. The
                ! diffference between the COBRA  normalization used here and the
                ! one used in COBRA is given by:
                !   - B_0 = sqrt(pres_0 mu_0) here
```

```
                !       versus
                !     B_0 = sqrt(2 pres_0 mu_0 / beta) in COBRA,
                !   - psi_0 = B_0 R_0^2 here
                !       versus
                !     psi_0 = B_0 (R_0/aspr)^2 in COBRA,
                ! This results in a difference in the factor T_0:
                !   - T_0 = sqrt(rho_0/pres_0) R_0 here
                !       versus
                !     T_0 = sqrt(rho_0/pres_0) R_0 sqrt(beta/2) in COBRA.
                ! Therefore, the Eigenvalues here should be rescaled as
                !   - EV_COBRA = EV_PB3D beta/2. This is done automatically.
                case (1)                                                ! MISHKA
                    ! user output
                    call writo('Using MISHKA normalization')

                    ! set the  major radius as the  average value of R_V  on the
                    ! magnetic axis
                    if (r_0.ge.huge(1._dp)) then                        ! user did not provide a value
                        r_0 = r_v_c(1,1,0)
                    else
                        nr_overridden_const = nr_overridden_const + 1
                    end if

                    ! set B_0 from magnetic field on axis
                    if (b_0.ge.huge(1._dp)) then                        ! user did not provide a value
                        b_0 = b_0_v
                    else
                        nr_overridden_const = nr_overridden_const + 1
                    end if

                    ! set reference pres_0 from B_0
                    if (pres_0.ge.huge(1._dp)) then                     ! user did not provide a value
                        pres_0 = b_0**2/mu_0_original
                    else
                        nr_overridden_const = nr_overridden_const + 1
                    end if

                    ! set reference flux from R_0 and B_0
                    if (psi_0.ge.huge(1._dp)) then                      ! user did not provide a value
                        psi_0 = r_0**2 * b_0
                    else
                        nr_overridden_const = nr_overridden_const + 1
                    end if
                case (2)                                                ! COBRA
                    ! user output
                    call writo('Using COBRA normalization')

                    ! set the major radius as the average geometric axis
                    if (r_0.ge.huge(1._dp)) then                        ! user did not provide a value
                        r_0 = 0.5_dp*(rmin_surf+rmax_surf)
                    else
                        nr_overridden_const = nr_overridden_const + 1
                    end if

                    ! set pres_0 from pressure on axis
                    if (pres_0.ge.huge(1._dp)) then                     ! user did not provide a value
                        pres_0 = pres_v(1,0)
                    else
                        nr_overridden_const = nr_overridden_const + 1
                    end if

                    ! set the reference value for B_0 from pres_0 and beta
                    if (b_0.ge.huge(1._dp)) then                        ! user did not provide a value
                        !B_0 = sqrt(2._dp*pres_0*mu_0_original/beta_V)  ! exact COBRA
                        b_0 = sqrt(pres_0*mu_0_original)                ! pure modified COBRA
                    else
                        nr_overridden_const = nr_overridden_const + 1
                    end if

                    ! set reference flux from R_0, B_0 and aspr
                    if (psi_0.ge.huge(1._dp)) then                      ! user did not provide a value
                        !psi_0 = B_0 * (R_0/aspr_V)**2                  ! exact COBRA
                        psi_0 = b_0 * r_0**2                            ! pure modified COBRA
                    else
                        nr_overridden_const = nr_overridden_const + 1
                    end if

                    ! user output concerning pure modified COBRA
                    call writo('Exact COBRA normalization is substituted by &
                        &"pure", modified version',warning=.true.)
                    call writo('This leaves the equations unmodified',&
                        &alert=.true.)
                    call writo('To translate to exact COBRA, manually multiply &
                        &PB3D Eigenvalues by',alert=.true.)
                    call lvl_ud(1)
                    call writo(trim(r2str(beta_v/2)),alert=.true.)
                    call lvl_ud(-1)
```

```
            end select

            ! rho_0 is set up through an input variable with the same name

            ! set Alfven time
            if (t_0.ge.huge(1._dp)) then                                    ! user did not provide a value
                t_0 = sqrt(mu_0_original*rho_0)*r_0/b_0
            else
                nr_overridden_const = nr_overridden_const + 1
            end if
        end subroutine calc_normalization_const_vmec

        ! HELENA version
        !> \private
        subroutine calc_normalization_const_hel
            ! user output
            call writo('Using MISHKA normalization')

            if (r_0.ge.huge(1._dp)) then                                    ! user did not provide a value
                r_0 = 1._dp
            else
                br_normalization_provided(1) = .true.
            end if
            if (b_0.ge.huge(1._dp)) then                                    ! user did not provide a value
                b_0 = 1._dp
            else
                br_normalization_provided(2) = .true.
            end if
            if (pres_0.ge.huge(1._dp)) then                                 ! user did not provide a value
                pres_0 = b_0**2/mu_0_original
            else
                nr_overridden_const = nr_overridden_const + 1
            end if
            if (psi_0.ge.huge(1._dp)) then                                  ! user did not provide a value
                psi_0 = r_0**2 * b_0
            else
                nr_overridden_const = nr_overridden_const + 1
            end if
            if (t_0.ge.huge(1._dp)) t_0 = sqrt(mu_0_original*rho_0)*r_0/b_0     ! only if user did not provide a value
        end subroutine calc_normalization_const_hel

        ! prints the Normalization factors
        !> \private
        subroutine print_normalization_const(R_0,rho_0,B_0,pres_0,psi_0,&
            &mu_0,T_0)
            ! input / output
            real(dp), intent(in), optional :: r_0
            real(dp), intent(in), optional :: rho_0
            real(dp), intent(in), optional :: b_0
            real(dp), intent(in), optional :: pres_0
            real(dp), intent(in), optional :: psi_0
            real(dp), intent(in), optional :: mu_0
            real(dp), intent(in), optional :: t_0

            ! user output
            if (present(r_0)) call writo('R_0    = '//trim(r2str(r_0))//' m')
            if (present(rho_0)) call writo('rho_0  = '//trim(r2str(rho_0))//&
                &' kg/m^3')
            if (present(b_0)) call writo('B_0    = '//trim(r2str(b_0))//' T')
            if (present(pres_0)) call writo('pres_0 = '//trim(r2str(pres_0))//&
                &' Pa')
            if (present(psi_0)) call writo('psi_0  = '//trim(r2str(psi_0))//&
                &' Tm^2')
            if (present(mu_0)) call writo('mu_0   = '//&
                &trim(r2str(mu_0))//' Tm/A')
            if (present(t_0)) call writo('T_0    = '//trim(r2str(t_0))//' s')
        end subroutine print_normalization_const
    end function calc_normalization_const

    !> Normalize input quantities.
    !!
    !! \see calc_normalization_const()
    subroutine normalize_input()
        use num_vars, only: use_normalization, eq_style, mu_0_original
        use vmec_ops, only: normalize_vmec
        use eq_vars, only: vac_perm

        ! only normalize if needed
        if (use_normalization) then
            ! user output
            call writo('Start normalizing the input variables')
            call lvl_ud(1)

            ! normalize common variables
            vac_perm = vac_perm/mu_0_original

            ! choose which equilibrium style is being used:
```

```
            !   1:  VMEC
            !   2:  HELENA
            select case (eq_style)
                case (1)                                                    ! VMEC
                    call normalize_vmec
                case (2)                                                    ! HELENA
                    ! other HELENA input already normalized
                    call writo('HELENA input is already normalized with MISHKA &
                        &normalization')
            end select

            ! user output
            call lvl_ud(-1)
            call writo('Normalization done')
        end if
    end subroutine normalize_input

    !> Plots the  magnetic fields.
    !!
    !! If multiple equilibrium parallel jobs, every  job does its piece, and the
    !! results are joined automatically by plot_HDF5.
    !!
    !! The outputs are  given in contra- and covariant  components and magnitude
    !! in multiple coordinate systems, as indicated in calc_vec_comp().
    !!
    !! The starting point is the fact that the magnetic field is given by
    !!  \f[\vec{B} = \frac{\vec{e}_{\theta}}{\mathcal{J}}, \f]
    !! in F coordinates. The F covariant components are therefore given by
    !!  \f[B_i = \frac{g_{i3}}{\mathcal{J}} =
    !!      \frac{\vec{e}_i \cdot \vec{e}_3}{\mathcal{J}}, \f]
    !! and the only non-vanishing contravariant component is
    !!  \f[B^3 = \frac{1}{\mathcal{J}}. \f]
    !!
    !! These are then all be transformed to the other coordinate systems.
    !!
    !! \note
    !!  -# Vector plots for different Richardson  levels can be combined to show
    !!  the total grid by just plotting them all individually.
    !!  -# The metric factors and transformation matrices have to be allocated.
    !!
    !! \return ierr
    integer function b_plot(grid_eq,eq_1,eq_2,rich_lvl,plot_fluxes,XYZ) &
        &result(ierr)

        use grid_utilities, only: calc_vec_comp
        use eq_utilities, only: calc_inv_met
        use num_vars, only: eq_style, norm_disc_prec_eq, use_normalization
        use num_utilities, only: c
        use eq_vars, only: b_0

        character(*), parameter :: rout_name = 'B_plot'

        ! input / output
        type(grid_type), intent(inout) :: grid_eq                           !< equilibrium grid
        type(eq_1_type), intent(in) :: eq_1                                 !< flux equilibrium variables
        type(eq_2_type), intent(in) :: eq_2                                 !< metric equilibrium variables
        integer, intent(in), optional :: rich_lvl                           !< Richardson level
        logical, intent(in), optional :: plot_fluxes                        !< plot the fluxes
        real(dp), intent(in), optional :: xyz(:,:,:,:)                      !< X, Y and Z of grid

        ! local variables
        integer :: id                                                       ! counter
        real(dp), allocatable :: b_com(:,:,:,:,:)                           ! covariant and contravariant components of
    B (dim1,dim2,dim3,3,2)
        real(dp), allocatable :: b_mag(:,:,:)                               ! magnitude of B (dim1,dim2,dim3)
        real(dp), allocatable, save :: b_flux_tor(:,:), b_flux_pol(:,:)     ! fluxes
        character(len=10) :: base_name                                      ! base name
        logical :: plot_fluxes_loc                                          ! local plot_fluxes

        ! initialize ierr
        ierr = 0

        ! tests
        if (eq_style.eq.1 .and. .not.allocated(grid_eq%trigon_factors)) then
            ierr = 1
            chckerr('trigonometric factors not allocated')
        end if

        ! set up local plot_fluxes
        plot_fluxes_loc = .false.
        if (present(plot_fluxes)) plot_fluxes_loc = plot_fluxes

        ! set up components and magnitude of B
        allocate(b_com(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r,3,2))
        allocate(b_mag(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
        b_com = 0._dp
        b_mag = 0._dp
```

```
        do id = 1,3
            b_com(:,:,:,id,1) = eq_2%g_FD(:,:,:,c([3,id],.true.),0,0,0)/&
                &eq_2%jac_FD(:,:,:,0,0,0)
        end do
        b_com(:,:,:,3,2) = 1._dp/eq_2%jac_FD(:,:,:,0,0,0)

        ! transform back to unnormalized quantity
        if (use_normalization) b_com = b_com * b_0

        ! set plot variables
        base_name = 'B'
        if (present(rich_lvl)) then
            if (rich_lvl.gt.0) base_name = trim(base_name)//'_R_'//&
                &trim(i2str(rich_lvl))
        end if

        ! transform coordinates, including the flux
        if (plot_fluxes_loc) then
            ierr = calc_vec_comp(grid_eq,grid_eq,eq_1,eq_2,b_com,&
                &norm_disc_prec_eq,v_mag=b_mag,base_name=base_name,&
                &v_flux_tor=b_flux_tor,v_flux_pol=b_flux_pol,xyz=xyz)
            chckerr('')
        else
            ierr = calc_vec_comp(grid_eq,grid_eq,eq_1,eq_2,b_com,&
                &norm_disc_prec_eq,v_mag=b_mag,base_name=base_name,xyz=xyz)
            chckerr('')
        end if
    end function b_plot

    !> Plots the current.
    !!
    !! If multiple equilibrium parallel jobs, every  job does its piece, and the
    !! results are joined automatically by plot_HDF5.
    !!
    !! The outputs are  given in contra- and covariant  components and magnitude
    !! in multiple coordinate systems, as indicated in calc_vec_comp().
    !!
    !! The starting point is the pressure balance
    !!  \f[ \nabla p = \vec{J} \times \vec{B}, \f]
    !! which, using \f$\vec{B} = \frac{\vec{e}_{\theta}}{\mathcal{J}}\f$,
    !! reduces to
    !!  \f[J^\alpha = -p'. \f]
    !! Furthermore, the current has to lie in the magnetic flux surfaces:
    !!  \f[J^\psi = 0. \f]
    !! Finally, the  parallel current \f$\sigma\f$  gives an expression  for the
    !! last contravariant component:
    !!  \f[J^\theta =
    !!      \frac{\sigma}{\mathcal{J}} + p' \frac{B_\alpha}{B_\theta}. \f]
    !! From these, the contravariant components can be calculated as
    !!  \f[J_i = J^\alpha g_{\alpha,i} + J^\theta g_{\theta,i}. \f]
    !!
    !! These are then all be transformed to the other coordinate systems.
    !! \note
    !!  -# Vector plots for different Richardson  levels can be combined to show
    !!  the total grid by just plotting them all individually.
    !!  -# The metric factors and transformation matrices have to be allocated.
    !!
    !! \return ierr
    integer function j_plot(grid_eq,eq_1,eq_2,rich_lvl,plot_fluxes,XYZ) &
        &result(ierr)

        use grid_utilities, only: calc_vec_comp
        use eq_utilities, only: calc_inv_met
        use num_vars, only: eq_style, norm_disc_prec_eq, use_normalization
        use num_utilities, only: c
        use eq_vars, only: b_0, r_0, pres_0
#if ldebug
        use eq_vars, only: max_flux_f
        use vmec_vars, only: j_v_sup_int
        use num_utilities, only: calc_int
#endif

        character(*), parameter :: rout_name = 'J_plot'

        ! input / output
        type(grid_type), intent(inout) :: grid_eq                           !< equilibrium grid
        type(eq_1_type), intent(in) :: eq_1                                 !< flux equilibrium variables
        type(eq_2_type), intent(in) :: eq_2                                 !< metric equilibrium variables
        integer, intent(in), optional :: rich_lvl                           !< Richardson level
        logical, intent(in), optional :: plot_fluxes                        !< plot the fluxes
        real(dp), intent(in), optional :: xyz(:,:,:,:)                      !< X, Y and Z of grid

        ! local variables
        integer :: id, kd                                                   ! counters
        real(dp), allocatable :: j_com(:,:,:,:,:)                           ! covariant and contravariant components of
    J (dim1,dim2,dim3,3,2)
        real(dp), allocatable :: j_mag(:,:,:)                               ! magnitude of J (dim1,dim2,dim3)
```

```
        character(len=10) :: base_name                                          ! base name
        real(dp), allocatable, save :: j_flux_tor(:,:), j_flux_pol(:,:)         ! fluxes
        logical :: plot_fluxes_loc                                              ! local plot_fluxes
        character(len=max_str_ln) :: plot_name                                  ! name of plot
        character(len=max_str_ln) :: plot_titles(2)                             ! titles of plot
#if ldebug
        real(dp), allocatable :: j_v_sup_int2(:,:)                              ! integrated J_V_sup_int
#endif

        ! initialize ierr
        ierr = 0

        ! tests
        if (eq_style.eq.1 .and. .not.allocated(grid_eq%trigon_factors)) then
            ierr = 1
            chckerr('trigonometric factors not allocated')
        end if

        ! set up local plot_fluxes
        plot_fluxes_loc = .false.
        if (present(plot_fluxes)) plot_fluxes_loc = plot_fluxes

        ! set up components and magnitude of J
        allocate(j_com(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r,3,2))
        allocate(j_mag(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
        j_com = 0._dp
        j_mag = 0._dp
#if ldebug
        if (debug_j_plot) then
            j_com(:,:,:,1,2) = eq_2%g_FD(:,:,:,c([3,3],.true.),0,1,0) - &
                &eq_2%g_FD(:,:,:,c([3,3],.true.),0,0,0)*&
                &eq_2%jac_FD(:,:,:,0,1,0)/&
                &eq_2%jac_FD(:,:,:,0,0,0) - &
                &eq_2%g_FD(:,:,:,c([2,3],.true.),0,0,1) + &
                &eq_2%g_FD(:,:,:,c([2,3],.true.),0,0,0)*&
                &eq_2%jac_FD(:,:,:,0,0,1)/&
                &eq_2%jac_FD(:,:,:,0,0,0)
            j_com(:,:,:,1,2) = j_com(:,:,:,1,2)/eq_2%jac_FD(:,:,:,0,0,0)**2
            j_com(:,:,:,3,2) = eq_2%g_FD(:,:,:,c([2,3],.true.),1,0,0) - &
                &eq_2%g_FD(:,:,:,c([2,3],.true.),0,0,0)*&
                &eq_2%jac_FD(:,:,:,1,0,0)/&
                &eq_2%jac_FD(:,:,:,0,0,0) - &
                &eq_2%g_FD(:,:,:,c([1,3],.true.),0,1,0) + &
                &eq_2%g_FD(:,:,:,c([1,3],.true.),0,0,0)*&
                &eq_2%jac_FD(:,:,:,0,1,0)/&
                &eq_2%jac_FD(:,:,:,0,0,0)
            j_com(:,:,:,3,2) = j_com(:,:,:,3,2)/eq_2%jac_FD(:,:,:,0,0,0)**2
        else
#endif
            do kd = 1,grid_eq%loc_n_r
                j_com(:,:,kd,1,2) = -eq_1%pres_FD(kd,1)
                j_com(:,:,kd,3,2) = eq_2%sigma(:,:,kd)/&
                    &eq_2%jac_FD(:,:,kd,0,0,0) + eq_1%pres_FD(kd,1)*&
                    &eq_2%g_FD(:,:,kd,c([1,3],.true.),0,0,0) / &
                    &eq_2%g_FD(:,:,kd,c([3,3],.true.),0,0,0)
            end do
#if ldebug
        end if
#endif
        do id = 1,3
            j_com(:,:,:,id,1) = &
                &j_com(:,:,:,1,2)*eq_2%g_FD(:,:,:,c([1,id],.true.),0,0,0) + &
                &j_com(:,:,:,3,2)*eq_2%g_FD(:,:,:,c([3,id],.true.),0,0,0)
        end do

        ! transform back to unnormalized quantity
        if (use_normalization) j_com = j_com * pres_0/(r_0*b_0)

        ! set plot variables
        base_name = 'J'
        if (present(rich_lvl)) then
            if (rich_lvl.gt.0) base_name = trim(base_name)//'_R_'//&
                &trim(i2str(rich_lvl))
        end if

        ! transform coordinates, including the flux
        if (plot_fluxes_loc) then
            ierr = calc_vec_comp(grid_eq,grid_eq,eq_1,eq_2,j_com,&
                &norm_disc_prec_eq,v_mag=j_mag,base_name=base_name,&
                &v_flux_tor=j_flux_tor,v_flux_pol=j_flux_pol,xyz=xyz)
            chckerr('')
        else
            ierr = calc_vec_comp(grid_eq,grid_eq,eq_1,eq_2,j_com,&
                &norm_disc_prec_eq,v_mag=j_mag,base_name=base_name,xyz=xyz)
            chckerr('')
        end if

```

```
#if ldebug
        if (eq_style.eq.1) then
            ! transform back to unnormalized quantity
            if (use_normalization) j_v_sup_int = j_v_sup_int * pres_0/(r_0*b_0)

            ! plot the fluxes from VMEC
            plot_name = 'J_V_sup_int'
            plot_titles = ['J^theta_V','J^phi_V  ']
            call print_ex_2d(plot_titles,plot_name,j_v_sup_int,&
                &x=reshape(grid_eq%r_F*2*pi/max_flux_f,&
                &[size(grid_eq%r_F),1]),draw=.false.)
            call draw_ex(plot_titles,plot_name,2,1,.false.)

            ! integrate
            allocate(j_v_sup_int2(size(j_v_sup_int,1),2))
            do id = 1,2
                ierr = calc_int(j_v_sup_int(:,id),grid_eq%r_E,&
                    &j_v_sup_int2(:,id))
                chckerr('')
            end do
            plot_name = 'J_V_sup_int2'
            plot_titles = ['J^theta_V','J^phi_V  ']
            call print_ex_2d(plot_titles,plot_name,j_v_sup_int2,&
                &x=reshape(grid_eq%r_F*2*pi/max_flux_f,&
                &[size(grid_eq%r_F),1]),draw=.false.)
            call draw_ex(plot_titles,plot_name,2,1,.false.)
        end if
#endif
    end function j_plot

    !> Plots the curvature.
    !!
    !! If multiple equilibrium parallel jobs, every  job does its piece, and the
    !! results are joined automatically by plot_HDF5.
    !!
    !! The outputs are  given in contra- and covariant  components and magnitude
    !! in multiple coordinate systems, as indicated in calc_vec_comp().
    !!
    !! The starting point is the curvature, given by
    !!  \f[\vec{\kappa} =
    !!      \kappa_n \frac{\nabla \psi}{ \left|\nabla \psi\right|^2 } +
    !!      \kappa_g \frac{\nabla \psi \times \vec{B}}{B^2} , \f]
    !! which can be  used to find the covariant and  contravariant components in
    !! Flux coordinates.
    !!
    !! These are then  transformed to Cartesian coordinates and plotted.
    !!
    !! \note
    !!  -# Vector plots for different Richardson  levels can be combined to show
    !!  the total grid by just plotting them all individually.
    !!  -# The metric factors and transformation matrices have to be allocated.
    !!
    !! \return ierr
    integer function kappa_plot(grid_eq,eq_1,eq_2,rich_lvl,XYZ) &
        &result(ierr)

        use grid_utilities, only: calc_vec_comp
        use eq_vars, only: eq_1_type, eq_2_type
        use eq_utilities, only: calc_inv_met
        use num_vars, only: eq_style, norm_disc_prec_eq, use_normalization
        use num_utilities, only: c
        use eq_vars, only: r_0
        use vmec_utilities, only: calc_trigon_factors
#if ldebug
        use num_vars, only: ltest, eq_jobs_lims, eq_job_nr
        use input_utilities, only: get_log
        use grid_utilities, only: trim_grid, calc_xyz_grid
#endif

        character(*), parameter :: rout_name = 'kappa_plot'

        ! input / output
        type(grid_type), intent(inout) :: grid_eq                               !< equilibrium grid
        type(eq_1_type), intent(in) :: eq_1                                     !< metric equilibrium variables
        type(eq_2_type), intent(in) :: eq_2                                     !< metric equilibrium variables
        integer, intent(in), optional :: rich_lvl                               !< Richardson level
        real(dp), intent(in), optional :: xyz(:,:,:,:)                          !< X, Y and Z of grid

        ! local variables
        real(dp), allocatable :: k_com(:,:,:,:,:)                               ! covariant and contravariant components of kappa (dim1,dim2,dim3,3,2)
        real(dp), allocatable :: k_mag(:,:,:)                                   ! magnitude of kappa (dim1,dim2,dim3)
        character(len=15) :: base_name                                          ! base name
#if ldebug
        type(grid_type) :: grid_trim                                            ! trimmed equilibrium grid
        integer :: id                                                           ! counter
        integer :: norm_id(2)                                                   ! untrimmed normal indices for trimmed grids
```

```
        integer :: plot_dim(4)                                          ! dimensions of plot
        integer :: plot_offset(4)                                       ! local offset of plot
        real(dp), allocatable :: xyz_loc(:,:,:,:,:)                     ! X, Y and Z of surface in cylindrical coordinates, trimmed grid
        real(dp), allocatable :: k_com_inv(:,:,:,:)                     ! inverted cartesian components of curvature
        logical, save :: asked_for_testing = .false.                    ! whether we have been asked to test
        logical, save :: testing = .false.                              ! whether we are testing
#endif

        ! initialize ierr
        ierr = 0

        ! tests
        if (eq_style.eq.1 .and. .not.allocated(grid_eq%trigon_factors)) then
            ierr = 1
            chckerr('trigonometric factors not allocated')
        end if

        ! set up components and magnitude of kappa
        allocate(k_com(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r,3,2))
        allocate(k_mag(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
        k_com = 0._dp
        k_mag = 0._dp

        ! covariant components:
        !   kappa . e_alpha = kappa_g
        !   kappa . e_psi   = kappa_n - kappa_g h12/h22
        !   kappa . e_theta = 0
        ! and similar for Q
        k_com(:,:,:,1,1) = eq_2%kappa_g
        k_com(:,:,:,2,1) = eq_2%kappa_n - &
            &eq_2%kappa_g * &
            &eq_2%h_FD(:,:,:,c([1,2],.true.),0,0,0)/&
            &eq_2%h_FD(:,:,:,c([2,2],.true.),0,0,0)
        k_com(:,:,:,3,1) = 0._dp

        ! contravariant components:
        !   kappa . nabla alpha = kappa_n h12 + kappa_g g33/(J^2 h22)
        !   kappa . nabla psi   = kappa_n h22
        !   kappa . nabla theta = kappa_n h23 - kappa_g g13/(J^2 h22)
        ! and similar for Q
        k_com(:,:,:,1,2) = eq_2%kappa_n * &
            &eq_2%h_FD(:,:,:,c([1,2],.true.),0,0,0) + &
            &eq_2%kappa_g * &
            &eq_2%g_FD(:,:,:,c([3,3],.true.),0,0,0) / &
            &(eq_2%jac_FD(:,:,:,0,0,0)**2*&
            &eq_2%h_FD(:,:,:,c([2,2],.true.),0,0,0))
        k_com(:,:,:,2,2) = eq_2%kappa_n * &
            &eq_2%h_FD(:,:,:,c([2,2],.true.),0,0,0)
        k_com(:,:,:,3,2) = eq_2%kappa_n * &
            &eq_2%h_FD(:,:,:,c([3,2],.true.),0,0,0) - &
            &eq_2%kappa_g * &
            &eq_2%g_FD(:,:,:,c([3,1],.true.),0,0,0) / &
            &(eq_2%jac_FD(:,:,:,0,0,0)**2*&
            &eq_2%h_FD(:,:,:,c([2,2],.true.),0,0,0))

        ! transform back to unnormalized quantity
        if (use_normalization) k_com = k_com / r_0

        ! set plot variables
        base_name = 'kappa'
        if (present(rich_lvl)) then
            if (rich_lvl.gt.0) base_name = trim(base_name)//'_R_'//&
                &trim(i2str(rich_lvl))
        end if

        ! transform coordinates
        ierr = calc_vec_comp(grid_eq,grid_eq,eq_1,eq_2,k_com,&
            &norm_disc_prec_eq,v_mag=k_mag,base_name=base_name,xyz=xyz)
        chckerr('')

#if ldebug
        if (ltest) then
            if (.not.asked_for_testing) then
                call writo('Do you want to plot the inversion in kappa?')
                call lvl_ud(1)
                testing = get_log(.false.)
                asked_for_testing = .true.
                call lvl_ud(-1)
            end if
            if (testing) then
                call writo('Every point in the grid is transformed by &
                    &displacing it in the direction of the curvature,')
                call writo('by a distance equal to the inverse of the &
                    &curvature. I.e. the point is displaced to the center &
                    &of curvature.')
                call lvl_ud(1)
```

```
                ! trim equilibrium grid
                ierr = trim_grid(grid_eq,grid_trim,norm_id)
                chckerr('')

                ! set up plot dimensions and local dimensions
                plot_dim = [grid_trim%n,3]
                plot_offset = [0,0,grid_trim%i_min-1,0]

                ! possibly modify if multiple equilibrium parallel jobs
                if (size(eq_jobs_lims,2).gt.1) then
                    plot_dim(1) = eq_jobs_lims(2,size(eq_jobs_lims,2)) - &
                        &eq_jobs_lims(1,1) + 1
                    plot_offset(1) = eq_jobs_lims(1,eq_job_nr) - 1
                end if

                ! set up inverted cartesian components
                allocate(k_com_inv(grid_trim%n(1),grid_trim%n(2),&
                    &grid_trim%loc_n_r,3))
                do id = 1,3
                    k_com_inv(:,:,:,id) = &
                        &k_com(:,:,norm_id(1):norm_id(2),id,1)/&
                        &(k_mag(:,:,norm_id(1):norm_id(2))**2)
                end do

                ! if VMEC, calculate trigonometric factors of trimmed grid
                if (eq_style.eq.1) then
                    ierr = calc_trigon_factors(grid_trim%theta_E,&
                        &grid_trim%zeta_E,grid_trim%trigon_factors)
                    chckerr('')
                end if

                ! calculate X, Y and Z
                allocate(xyz_loc(grid_trim%n(1),grid_trim%n(2),&
                    &grid_trim%loc_n_r,3,3))
                ierr = calc_xyz_grid(grid_eq,grid_trim,xyz_loc(:,:,:,1,1),&
                    &xyz_loc(:,:,:,1,2),xyz_loc(:,:,:,1,3))
                chckerr('')

                ! produce a plot of center of curvature for every point
                call plot_hdf5(['cen_of_curv'],'TEST_cen_of_curv_vec',&
                    &k_com_inv,tot_dim=plot_dim,loc_offset=plot_offset,&
                    &x=xyz_loc(:,:,:,:,1),y=xyz_loc(:,:,:,:,2),&
                    &z=xyz_loc(:,:,:,:,3),col=4,cont_plot=eq_job_nr.gt.1,&
                    &descr='center of curvature')

                ! displace the points to the center of curvature
                do id = 1,3
                    xyz_loc(:,:,:,1,id) = xyz_loc(:,:,:,1,id) + &
                        &k_com_inv(:,:,:,id)
                end do
                xyz_loc(:,:,:,2,:) = xyz_loc(:,:,:,1,:)
                xyz_loc(:,:,:,3,:) = xyz_loc(:,:,:,3,:)

                ! produce an inverted plot
                call plot_hdf5(['cen_of_curv_inv'],'TEST_cen_of_curv_inv_vec',&
                    &-k_com_inv,tot_dim=plot_dim,loc_offset=plot_offset,&
                    &x=xyz_loc(:,:,:,:,1),y=xyz_loc(:,:,:,:,2),&
                    &z=xyz_loc(:,:,:,:,3),col=4,cont_plot=eq_job_nr.gt.1,&
                    &descr='center of curvature')

                ! clean up
                call grid_trim%dealloc()

                call lvl_ud(-1)
            end if
        end if
#endif
    end function kappa_plot

    !> Plots  \b HALF  of the  change in  the position  vectors for  2 different
    !! toroidal positions, which can correspond to a ripple.
    !!
    !! Also calculates \b HALF of the relative magnetic perturbation, which also
    !! corresponds to a ripple.
    !!
    !! Finally, if the  output grid contains a  fundamental interval \f$2\pi\f$,
    !! the proportionality between both is written to a file.
    !!
    !! \note  The  metric   factors  and  transformation  matrices  have  to  be
    !! allocated.
    !!
    !! \return ierr
    integer function delta_r_plot(grid_eq,eq_1,eq_2,XYZ,rich_lvl) &
        &result(ierr)

        use grid_utilities, only: calc_vec_comp, calc_xyz_grid, trim_grid, &
```

```
            &calc_tor_diff, nufft
        use eq_utilities, only: calc_inv_met
        use num_vars, only: eq_style, norm_disc_prec_eq, eq_job_nr, rz_0, &
            &use_normalization, rank, ex_plot_style, n_procs, prop_b_tor_i, &
            &tol_zero
        use eq_vars, only: b_0
        use num_utilities, only: c, calc_int, order_per_fun
        use mpi_utilities, only: get_ser_var

        character(*), parameter :: rout_name = 'delta_r_plot'

        ! input / output
        type(grid_type), intent(inout) :: grid_eq                          !< equilibrium grid
        type(eq_1_type), intent(in) :: eq_1                                !< flux equilibrium variables
        type(eq_2_type), intent(in) :: eq_2                                !< metric equilibrium variables
        real(dp), intent(in) :: xyz(:,:,:,:)                               !< X, Y and Z of grid
        integer, intent(in), optional :: rich_lvl                          !< Richardson level

        ! local variables
        integer :: id, kd                                                  ! counters
        integer :: norm_id(2)                                              ! untrimmed normal indices for trimmed grids
        integer :: r_lim_2d(2)                                             ! limits of r to plot in 2-D
        integer :: max_m                                                   ! maximum mode number
        real(dp), allocatable :: xyz_loc(:,:,:,:)                          ! X, Y and Z of surface in trimmed grid
        real(dp), allocatable :: b_com(:,:,:,:,:)                          ! covariant and contravariant components of B (dim1,dim2,dim3,3,2)
        real(dp), allocatable :: delta_b_tor(:,:,:)                        ! delta_B/B
        real(dp), allocatable :: prop_b_tor_tot(:,:)                       ! delta_r / delta_B/B in total grid
        real(dp), allocatable :: prop_b_tor(:,:,:)                         ! delta_r / delta_B/B
        real(dp), allocatable :: var_tot_loc(:)                            ! auxilliary variable
        real(dp), allocatable :: x_2d(:,:)                                 ! x for 2-D plotting
        real(dp), allocatable :: delta_r(:,:,:)                            ! normal displacement
        real(dp), allocatable :: theta_geo(:,:,:)                          ! geometrical poloidal angle for proportionality factor output
        real(dp), allocatable :: r_geo(:,:,:)                              ! geometrical radius for proportionality factor output
        real(dp), allocatable :: prop_b_tor_plot(:,:)                      ! prop_B_tor and angle at last normal position for plotting
        real(dp), allocatable :: delta_r_f(:,:)                            ! Fourier components of delta_r at last normal position
        logical :: new_file_found                                          ! name for new file found
        character(len=25) :: base_name                                     ! base name
        character(len=25) :: plot_name                                     ! plot name
        character(len=max_str_ln) :: prop_b_tor_file_name                  ! name of B_tor proportionality file
        type(grid_type) :: grid_trim                                       ! trimmed equilibrium grid

        ! initialize ierr
        ierr = 0

        ! tests
        if (eq_style.eq.1 .and. .not.allocated(grid_eq%trigon_factors)) then
            ierr = 1
            chckerr('trigonometric factors not allocated')
        end if

        call writo('Calculate normal displacement')
        call lvl_ud(1)

        ! trim grid
        ierr = trim_grid(grid_eq,grid_trim,norm_id)
        chckerr('')

        ! calculate local X, Y and Z
        ! (Can be different from provided one for different plot grid styles)
        allocate(xyz_loc(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r,4))
        ierr = calc_xyz_grid(grid_eq,grid_eq,xyz_loc(:,:,:,1),xyz_loc(:,:,:,2),&
            &xyz_loc(:,:,:,3),r=xyz_loc(:,:,:,4))
        chckerr('')

        ! calculate geometrical poloidal angle and radius
        allocate(theta_geo(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
        allocate(r_geo(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
        theta_geo = atan2(xyz_loc(:,:,:,3)-rz_0(2),&
            &xyz_loc(:,:,:,4)-rz_0(1))
        where (theta_geo.lt.0._dp) theta_geo = theta_geo + 2*pi
        r_geo = sqrt((xyz_loc(:,:,:,3)-rz_0(2))**2 + &
            &(xyz_loc(:,:,:,4)-rz_0(1))**2)

        ! plot
        call plot_hdf5('theta_geo','theta_geo',&
            &theta_geo(:,:,norm_id(1):norm_id(2)),&
            &tot_dim=[grid_trim%n(1),grid_trim%n(2),grid_trim%n(3)],&
            &loc_offset=[0,0,grid_trim%i_min-1],&
            &x=xyz(:,:,norm_id(1):norm_id(2),1),&
            &y=xyz(:,:,norm_id(1):norm_id(2),2),&
            &z=xyz(:,:,norm_id(1):norm_id(2),3),cont_plot=eq_job_nr.gt.1,&
            &descr='geometrical poloidal angle')
```

```
        call plot_hdf5('r_geo','r_geo',&
            &r_geo(:,:,norm_id(1):norm_id(2)),&
            &tot_dim=[grid_trim%n(1),grid_trim%n(2),grid_trim%n(3)],&
            &loc_offset=[0,0,grid_trim%i_min-1],&
            &x=xyz(:,:,norm_id(1):norm_id(2),1),&
            &y=xyz(:,:,norm_id(1):norm_id(2),2),&
            &z=xyz(:,:,norm_id(1):norm_id(2),3),cont_plot=eq_job_nr.gt.1,&
            &descr='geometrical radius')

        ! calculate perturbation delta_r on radius
        ierr = calc_tor_diff(r_geo,theta_geo,norm_disc_prec_eq,&
            &absolute=.true.,r=grid_eq%loc_r_F)
        chckerr('')
        allocate(delta_r(grid_eq%n(1),1,grid_eq%loc_n_r))
        delta_r(:,1,:) = r_geo(:,2,:)*0.5_dp                                    ! take factor half as this is absolute
        deallocate(r_geo)

        ! set plot variables for delta_r
        base_name = 'delta_r'
        plot_name = 'delta_r'
        if (present(rich_lvl)) then
            if (rich_lvl.gt.0) base_name = trim(base_name)//'_R_'//&
                &trim(i2str(rich_lvl))
        end if

        ! plot
        call plot_hdf5(trim(plot_name),trim(base_name),&
            &delta_r(:,:,norm_id(1):norm_id(2)),&
            &tot_dim=[grid_trim%n(1),1,grid_trim%n(3)],&
            &loc_offset=[0,0,grid_trim%i_min-1],&
            &x=xyz(:,2:2,norm_id(1):norm_id(2),1),&
            &y=xyz(:,2:2,norm_id(1):norm_id(2),2),&
            &z=xyz(:,2:2,norm_id(1):norm_id(2),3),&
            &cont_plot=eq_job_nr.gt.1,&
            &descr='plasma position displacement')

        ! calculate NUFFT for last point
        if (rank.eq.n_procs-1) then
            ! plot delta_r at last normal position
            call print_ex_2d(trim(plot_name),trim(base_name),&
                &delta_r(1:grid_eq%n(1)-1,1,grid_eq%loc_n_r),&
                &x=theta_geo(1:grid_eq%n(1)-1,2,grid_eq%loc_n_r),&
                &draw=.false.)
            call draw_ex([trim(plot_name)],trim(base_name),1,1,.false.)

            ! calculate and plot Fourier modes
            ierr = nufft(theta_geo(1:grid_eq%n(1)-1,2,grid_eq%loc_n_r),&
                &delta_r(1:grid_eq%n(1)-1,1,grid_eq%loc_n_r),delta_r_f)
            chckerr('')
            base_name = trim(base_name)//'_F'
            call print_ex_2d(['delta_r cos','delta_r sin'],trim(base_name),&
                &delta_r_f,draw=.false.)
            call draw_ex(['delta_r cos','delta_r sin'],trim(base_name),2,1,&
                &.false.)

            ! mode outputs, (p)recycle "prop_B_tor_file_*"
            new_file_found = .false.
            prop_b_tor_file_name = base_name
            kd = 1
            do while (.not.new_file_found)
                open(prop_b_tor_i,file=trim(prop_b_tor_file_name)//'_'//&
                    &trim(i2str(kd))//'.dat',iostat=ierr,status='old')
                if (ierr.eq.0) then
                    kd = kd + 1
                else
                    prop_b_tor_file_name = trim(prop_b_tor_file_name)//&
                        &'_'//trim(i2str(kd))//'.dat'
                    new_file_found = .true.
                    open(prop_b_tor_i,file=trim(prop_b_tor_file_name),&
                        &iostat=ierr,status='new')
                    chckerr('Failed to open file')
                end if
            end do

            ! write to output file
            max_m = 20
            write(prop_b_tor_i,'("# ",2(A5," "),2(A23," "))') &
                &'N', 'M', 'delta_c', 'delta_s'
            do id = 1,min(size(delta_r_f,1),max_m+1)
                write(prop_b_tor_i,'("  ",2(I5," "),2(ES23.16," "))') &
                    &18, id-1, delta_r_f(id,:)
            end do

            ! close
            close(prop_b_tor_i)
        end if
```

```
        call lvl_ud(-1)

        call writo('Calculate magnetic field ripple')
        call lvl_ud(1)

        ! Calculate cylindrical components of B
        allocate(b_com(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r,3,2))
        b_com = 0._dp
        do id = 1,3
            b_com(:,:,:,id,1) = eq_2%g_FD(:,:,:,c([3,id],.true.),0,0,0)/&
                &eq_2%jac_FD(:,:,:,0,0,0)
        end do
        b_com(:,:,:,3,2) = 1._dp/eq_2%jac_FD(:,:,:,0,0,0)
        if (use_normalization) b_com = b_com * b_0
        ierr = calc_vec_comp(grid_eq,grid_eq,eq_1,eq_2,b_com,norm_disc_prec_eq,&
            &max_transf=4)
        chckerr('')

        ! calculate perturbation
        ierr = calc_tor_diff(b_com,theta_geo,norm_disc_prec_eq,&
            &r=grid_eq%loc_r_F)
        chckerr('')
        allocate(delta_b_tor(grid_eq%n(1),1,grid_eq%loc_n_r))
        delta_b_tor(:,1,:) = 0.5_dp*sum(b_com(:,2,:,2,:),3)
        deallocate(b_com)

        ! set plot variables for delta_B_tor
        base_name = 'delta_B_tor'
        plot_name = 'delta_B_tor'
        if (present(rich_lvl)) then
            if (rich_lvl.gt.0) base_name = trim(base_name)//'_R_'//&
                &trim(i2str(rich_lvl))
        end if

        ! plot with HDF5
        call plot_hdf5(trim(plot_name),trim(base_name),&
            &delta_b_tor(:,:,norm_id(1):norm_id(2)),&
            &tot_dim=[grid_trim%n(1),1,grid_trim%n(3)],&
            &loc_offset=[0,0,grid_trim%i_min-1],&
            &x=xyz(:,2:2,norm_id(1):norm_id(2),1),&
            &y=xyz(:,2:2,norm_id(1):norm_id(2),2),&
            &z=xyz(:,2:2,norm_id(1):norm_id(2),3),cont_plot=eq_job_nr.gt.1,&
            &descr='plasma position displacement')

        call lvl_ud(-1)

        call writo('Calculate proportionality factor')
        call lvl_ud(1)

        allocate(prop_b_tor(grid_eq%n(1),1,grid_eq%loc_n_r))
        prop_b_tor = 0.0_dp
        where (abs(delta_b_tor).gt.tol_zero) prop_b_tor = delta_r / delta_b_tor

        ! set plot variables for prop_B_tor
        base_name = 'prop_B_tor'
        plot_name = trim(base_name)
        if (present(rich_lvl)) then
            if (rich_lvl.gt.0) base_name = trim(base_name)//'_R_'//&
                &trim(i2str(rich_lvl))
        end if

        ! plot with HDF5
        call plot_hdf5(plot_name,trim(base_name),&
            &prop_b_tor(:,:,norm_id(1):norm_id(2)),&
            &tot_dim=[grid_trim%n(1),1,grid_trim%n(3)],&
            &loc_offset=[0,0,grid_trim%i_min-1],&
            &x=xyz(:,2:2,norm_id(1):norm_id(2),1),&
            &y=xyz(:,2:2,norm_id(1):norm_id(2),2),&
            &z=xyz(:,2:2,norm_id(1):norm_id(2),3),&
            &cont_plot=eq_job_nr.gt.1,&
            &descr='delta_r divided by delta_B_tor')

        ! plot in 2-D
        r_lim_2d = [1,grid_trim%n(3)]
        if (ex_plot_style.eq.2) r_lim_2d(1) = &
            &max(r_lim_2d(1),r_lim_2d(2)-127+1)                    ! Bokeh can only handle 255/2 input arguments
        if (rank.eq.0) then
            allocate(x_2d(grid_trim%n(1),grid_trim%n(3)))
        end if
        allocate(prop_b_tor_tot(grid_trim%n(1),grid_trim%n(3)))
        do id = 1,grid_trim%n(1)
            ! prop_B_tor (all procs)
            ierr = get_ser_var(prop_b_tor(id,1,norm_id(1):norm_id(2)),&
                &var_tot_loc,scatter=.true.)
            chckerr('')
            prop_b_tor_tot(id,:) = max(min(var_tot_loc,2._dp),-2._dp)          ! limit to avoid division by small
```

delta_B_tor values

```
            deallocate(var_tot_loc)

            ! theta_geo (only master)
            ierr = get_ser_var(theta_geo(id,2,norm_id(1):norm_id(2)),&
                &var_tot_loc)
            chckerr('')
            if (rank.eq.0) x_2d(id,:) = var_tot_loc/pi
            deallocate(var_tot_loc)
        end do
        if (rank.eq.0) then
            call print_ex_2d([trim(plot_name)],trim(base_name),&
                &prop_b_tor_tot(:,r_lim_2d(1):r_lim_2d(2)),&
                &x=x_2d(:,r_lim_2d(1):r_lim_2d(2)),draw=.false.)
            call draw_ex([trim(plot_name)],trim(base_name),&
                &r_lim_2d(2)-r_lim_2d(1)+1,1,.false.)
        end if

        call lvl_ud(-1)

        call writo('Output to file as function of poloidal flux angle')
        call lvl_ud(1)

        ! only last rank outputs to file (it has the last normal position)
        if (rank.eq.n_procs-1) then
            ! find new file name
            new_file_found = .false.
            prop_b_tor_file_name = 'prop_B_tor'
            kd = 1
            do while (.not.new_file_found)
                open(prop_b_tor_i,file=trim(prop_b_tor_file_name)//'_'//&
                    &trim(i2str(kd))//'.dat',iostat=ierr,status='old')
                if (ierr.eq.0) then
                    kd = kd + 1
                else
                    prop_b_tor_file_name = trim(prop_b_tor_file_name)//&
                        &'_'//trim(i2str(kd))//'.dat'
                    new_file_found = .true.
                    open(prop_b_tor_i,file=trim(prop_b_tor_file_name),&
                        &iostat=ierr,status='new')
                    chckerr('Failed to open file')
                end if
            end do

            ! user output
            call writo('Save toroidal field proportionality factor in &
                &file "'//trim(prop_b_tor_file_name)//'"',persistent=.true.)

            ! order  output, taking away last  point as it is  equivalent to
            ! the first
            ierr = order_per_fun(reshape([&
                &theta_geo(1:grid_trim%n(1)-1,1,grid_trim%loc_n_r),&
                &prop_b_tor_tot(1:grid_trim%n(1)-1,grid_trim%n(3))],&
                &[grid_trim%n(1)-1,2]),prop_b_tor_plot,0)
            chckerr('')

            ! write to output file
            write(prop_b_tor_i,'("# ",2(A23," "))') &
                &'pol. angle [ ]', 'prop. factor [ ]'
            do id = 1,grid_trim%n(1)-1
                write(prop_b_tor_i,'("  ",2(ES23.16," "))') &
                    &prop_b_tor_plot(id,:)
            end do

            ! close
            close(prop_b_tor_i)
        end if

        call lvl_ud(-1)

        ! clean up
        call grid_trim%dealloc()
    end function delta_r_plot

    !> Divides the equilibrium jobs.
    !!
    !! For PB3D,  the entire  parallel range  has to be  calculated, but  due to
    !! memory limits this  has to be split  up in pieces. Every piece  has to be
    !! able to  contain the equilibrium variables  (see note below), as  well as
    !! the  vectorial  perturbation variables.  These  are  later combined  into
    !! tensorial variables and integrated.
    !!
    !! The  equilibrium variables  have to  be  operated on  to calculate  them,
    !! which  translates to  a scale  factor \c  mem_scale_fac. However,  in the
    !! perturbation phase,  when they are  just used,  this scale factor  is not
    !! needed.
    !!
```

```
    !! In its most  extreme form, the division in equilibrium  jobs would be the
    !! individual  calculation  on a  fundamental  integration  integral of  the
    !! parallel points:
    !!  - for \c magn_int_style=1 (trapezoidal), this is 1 point,
    !!  - for \c magn_int_style=2 (Simpson 3/8), this is 3 points.
    !!
    !! For  HELENA,  the parallel  derivatives  are  calculated discretely,  the
    !! equilibrium and  vectorial perturbation variables are  tabulated first in
    !! this HELENA  grid. This  happens in  the first  Richardson level.  In all
    !! Richardson levels,  afterwards, these  variables are interpolated  in the
    !! angular directions. In this case, therefore,  there can be no division of
    !! this HELENA output interval for the first Richardson level.
    !!
    !! This procedure  does the  job of  dividing the  grids setting  the global
    !! variables \c eq_jobs_lims.
    !!
    !! The integration of the tensorial perturbation variables is adjusted:
    !!  - If  the first job  of the parallel jobs  and not the  first Richardson
    !!  level: add half  of the integrated tensorial  perturbation quantities of
    !!  the previous level.
    !!  -  If  not the  first  job  of the  parallel  jobs,  add the  integrated
    !!  tensorial perturbation quantities to those of the previous parallel job,
    !!  same Richardson level.
    !!
    !! In fact,  the equilibrium jobs  have much  in common with  the Richardson
    !! levels,  as is  attested by  the existence  of the  routines do_eq()  and
    !! eq_info(), which are equivalent to do_rich() and rich_info().
    !!
    !! In  POST, finally,  the situation  is slightly  different for  HELENA, as
    !! all  the requested  variables  have to  fit,  including the  interpolated
    !! variables, as they are stored whereas  in PB3D they are not. The parallel
    !! range to be  taken is then the  one of the output grid,  including a base
    !! range for the variables tabulated on  the HELENA grid. Also, for extended
    !! output grids,  the size  of the  grid in  the secondary  angle has  to be
    !! included in \c n_par_X (i.e. toroidal when poloidal flux is used and vice
    !! versa). Furthermore, multiple equilibrium jobs are allowed.
    !!
    !! To this  end, optionally, a base  number can be provided  for \c n_par_X,
    !! that is always added to the number of points in the divided \c n_par_X.
    !!
    !! \note For  PB3D, only the  variables \c g_FD, \c  h_FD and \c  jac_FD are
    !! counted, as the equilibrium variables and the transformation matrices are
    !! deleted after use. Also, \c S, \c sigma, \c kappa_n and \c kappa_g can be
    !! neglected  as they  do not  contain  derivatives and  are therefore  much
    !! smaller. in both routines  calc_memory_eq() and calc_memory_x(), however,
    !! a 50% safety factor is used to account for this somewhat.
    !!
    !! \return ierr
    integer function divide_eq_jobs(n_par_X,arr_size,n_div,n_div_max,&
        &n_par_X_base,range_name) result(ierr)

        use num_vars, only: max_tot_mem, max_x_mem, magn_int_style, &
            &mem_scale_fac, prog_style
        use rich_vars, only: rich_lvl
        use eq_utilities, only: calc_memory_eq
        use x_utilities, only: calc_memory_x
        use x_vars, only: n_mod_x

        character(*), parameter :: rout_name = 'divide_eq_jobs'

        ! input / output
        integer, intent(in) :: n_par_x                                  !< number of parallel points to be divided
        integer, intent(in) :: arr_size(2)                              !< array size (using loc_n_r) for eq_2 and X_1 variables
        integer, intent(inout) :: n_div                                 !< final number of divisions
        integer, intent(in), optional :: n_div_max                      !< maximum n_div
        integer, intent(in), optional :: n_par_x_base                   !< base n_par_X, undivisible
        character(len=*), intent(in), optional :: range_name            !< name of range

        ! local variables
        real(dp) :: mem_size(2)                                         ! approximation of memory required for eq_2 and X_1 variables
        integer :: n_par_x_base_loc                                     ! local n_par_X_base
        integer :: max_mem_req                                          ! maximum memory required
        integer :: n_div_max_loc                                        ! maximum n_div
        integer :: n_par_range                                          ! nr. of points in range
        character(len=max_str_ln) :: range_message                      ! message about how many ranges
        character(len=max_str_ln) :: err_msg                            ! error message
        character(len=max_str_ln) :: range_name_loc                     ! local range name

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Dividing the equilibrium jobs')
        call lvl_ud(1)

```

```fortran
        ! set up width of fundamental interval
        select case (magn_int_style)
            case (1)                                                    ! Trapezoidal rule
                fund_n_par = 1
            case (2)                                                    ! Simpson's 3/8 rule
                fund_n_par = 3
        end select

        ! set up local n_par_X_base
        n_par_x_base_loc = 0
        if (present(n_par_x_base)) n_par_x_base_loc = n_par_x_base

        ! set up local range name
        range_name_loc = 'parallel points'
        if (present(range_name)) range_name_loc = trim(range_name)

        ! setup auxilliary variables
        n_div = 0
        mem_size = huge(1._dp)*0.49_dp
        if (rich_lvl.eq.1) then
            n_div_max_loc = (n_par_x-1)/fund_n_par
        else
            n_div_max_loc = n_par_x/fund_n_par
        end if
        if (present(n_div_max)) n_div_max_loc = n_div_max
        n_div_max_loc = max(1,n_div_max_loc)                            ! cannot have less than 1 piece

        ! calculate largest possible range of parallel points fitting in memory
        do while (max_tot_mem.lt.sum(mem_size))                         ! combined mem_size has to be smaller than total memory
            n_div = n_div + 1
            n_par_range = ceiling(n_par_x*1._dp/n_div + n_par_x_base_loc)
            ierr = calc_memory_eq(arr_size(1),n_par_range,mem_size(1))
            chckerr('')
            mem_size(1) = mem_size(1)*mem_scale_fac                     ! operations have to be done on eq, it is not just stored.
            ierr = calc_memory_x(1,arr_size(2)*n_par_range,n_mod_x,mem_size(2)) ! all modes have to be stored
            chckerr('')
            if (n_div.gt.n_div_max_loc) then                            ! still not enough memory
                ierr = 1
                err_msg = 'The memory limit is too low, need more than '//&
                    &trim(i2str(max_mem_req))//'MB'
                chckerr(err_msg)
            end if
            max_mem_req = ceiling(sum(mem_size))
        end do
        if (n_div.gt.1) then
            range_message = 'The '//trim(i2str(n_par_x))//' '//&
                &trim(range_name_loc)//' are split into '//&
                &trim(i2str(n_div))//' and '//trim(i2str(n_div))//&
                &' collective jobs are done serially'
        else
            range_message = 'The '//trim(i2str(n_par_x))//' '//&
                &trim(range_name_loc)//' can be done without splitting them'
        end if
        call writo(range_message)
        call writo('The total memory for all processes together is estimated &
            &to be about '//trim(i2str(ceiling(sum(mem_size))))//'MB')
        call lvl_ud(1)
        call writo('(maximum: '//trim(i2str(ceiling(max_tot_mem)))//'&
            &MB, user specified)')
        call lvl_ud(-1)

        if (prog_style.eq.1) then
            ! calculate max memory available for perturbation calculations
            mem_size(1) = mem_size(1)/mem_scale_fac                     ! rescale by mem_scale_fac as eq is now just stored
            max_x_mem = max_tot_mem - sum(mem_size)
            call writo('In the perturbation phase, the equilibrium variables &
                &are not being operated on:')
            call lvl_ud(1)
            call writo('This translates to a scale factor 1/'//&
                &trim(r2strt(mem_scale_fac)))
            call writo('Therefore, the memory left for the perturbation phase &
                &is '//trim(i2str(ceiling(max_x_mem)))//'MB')
            call lvl_ud(-1)
        end if

        ! user output
        call lvl_ud(-1)
        call writo('Equilibrium jobs divided')
    end function divide_eq_jobs

    !> Calculate \c eq_jobs_lims.
    !!
    !! Take into account that every job has to start and end at the
    !! start and end of a fundamental integration interval, as discussed in
```

```
    !! divide_eq_jobs():
    !!  - for \c magn_int_style=1 (trapezoidal), this is 1 point,
    !!  - for \c magn_int_style=2 (Simpson 3/8), this is 3 points.
    !!
    !! for POST, there are no Richardson levels,  and there has to be overlap of
    !! one always, in order to have correct composite integrals of the different
    !! regions.
    !!
    !! \note The  \c n_par_X passed into  this procedure refers to  the quantity
    !! that is already possibly halved if the Richardson level is higher than 1.
    !! This  information is  then  reflected in  the  eq_jobs_lims, which  refer
    !! to  the local  limits,  i.e.  only the  parallel  points currently  under
    !! consideration.
    !!
    !! \return ierr
    integer function calc_eq_jobs_lims(n_par_X,n_div) result(ierr)
        use num_vars, only: prog_style, eq_jobs_lims, eq_job_nr
        use rich_vars, only: rich_lvl

        character(*), parameter :: rout_name = 'calc_eq_jobs_lims'

        ! input / output
        integer, intent(in) :: n_par_x                                          !< number of parallel points in this Richardson level
        integer, intent(in) :: n_div                                            !< nr. of divisions of parallel ranges

        ! local variables
        integer :: id                                                           ! counter
        integer :: ol_width                                                     ! overlap width
        integer, allocatable :: n_par(:)                                        ! number of points per range
        character(len=max_str_ln) :: err_msg                                    ! error message

        ! initialize ierr
        ierr = 0

        ! Also initialize eq_job_nr to 0 as it is incremented by "do_eq".
        eq_job_nr = 0

        allocate(n_par(n_div))
        n_par = n_par_x/n_div                                                   ! number of parallel points on this processor
        n_par(1:mod(n_par_x,n_div)) = n_par(1:mod(n_par_x,n_div)) + 1           ! add a point to if there is a remainder
        chckerr('')

        ! (re)allocate equilibrium jobs limits
        if (allocated(eq_jobs_lims)) deallocate(eq_jobs_lims)
        allocate(eq_jobs_lims(2,n_div))

        ! set overlap width
        select case (prog_style)
            case (1)                                                            ! PB3D
                if (rich_lvl.eq.1) then
                    ol_width = 1
                else
                    ol_width = 0
                end if
            case (2)                                                            ! POST
                ol_width = 1
        end select

        ! loop over divisions
        do id = 1,n_div
            ! setup  first  guess,  without   taking  into  account  fundamental
            ! integration intervals
            if (id.eq.1) then
                eq_jobs_lims(1,id) = 1
            else
                eq_jobs_lims(1,id) = eq_jobs_lims(2,id-1) + 1 - ol_width
            end if
            eq_jobs_lims(2,id) = sum(n_par(1:id))

            ! take into account fundamental interval
            if (n_div.gt.1) then
                eq_jobs_lims(2,id) = eq_jobs_lims(1,id) - 1 + ol_width + &
                    &fund_n_par * max(1,nint(&
                    &(eq_jobs_lims(2,id)-eq_jobs_lims(1,id)+1._dp-ol_width)/&
                    &fund_n_par))
            end if
        end do

        ! test whether end coincides with sum of n_par
        if (eq_jobs_lims(2,n_div).ne.sum(n_par)) then
            ierr = 1
            err_msg = 'Limits don''t match, try with more memory or lower &
                &magn_int_style'
            chckerr(err_msg)
        end if
```

```
    end function calc_eq_jobs_lims

#if ldebug
    !> See if \c T_EF it complies with the theory of \cite Weyens3D.
    !!
    !! \note Debug version only
    !!
    !! \return ierr
    integer function test_t_ef(grid_eq,eq_1,eq_2) result(ierr)
        use num_vars, only: use_pol_flux_f, eq_style
        use grid_utilities, only: trim_grid
        use num_utilities, only: c
        use output_ops, only: plot_diff_hdf5

        character(*), parameter :: rout_name = 'test_T_EF'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(eq_1_type), intent(in) :: eq_1                                     !< flux equilibrium
        type(eq_2_type), intent(in) :: eq_2                                     !< metric equilibrium

        ! local variables
        integer :: id, kd                                                       ! counter
        integer :: norm_id(2)                                                   ! untrimmed normal indices for trimmed grids
        real(dp), allocatable :: res(:,:,:,:)                                   ! calculated result
        character(len=max_str_ln) :: file_name                                  ! name of plot file
        character(len=max_str_ln) :: description                                ! description of plot
        integer :: tot_dim(3), loc_offset(3)                                    ! total dimensions and local offset
        type(grid_type) :: grid_trim                                            ! trimmed equilibrium grid

        ! initialize ierr
        ierr = 0

        ! output
        call writo('Going to test whether T_EF complies with the theory')
        call lvl_ud(1)

        ! trim extended grid into plot grid
        ierr = trim_grid(grid_eq,grid_trim,norm_id)
        chckerr('')

        ! set total and local dimensions and local offset
        tot_dim = [grid_trim%n(1),grid_trim%n(2),grid_trim%n(3)]
        loc_offset = [0,0,grid_trim%i_min-1]

        ! set up res
        allocate(res(grid_trim%n(1),grid_trim%n(2),grid_trim%loc_n_r,9))
        res = 0.0_dp

        ! calc  res,  depending  on flux  used  in F  coordinates  and on  which
        ! equilibrium style is being used:
        !   1:  VMEC
        !   2:  HELENA
        select case (eq_style)
            case (1)                                                            ! VMEC
                if (use_pol_flux_f) then                                        ! using poloidal flux
                    ! calculate T_EF(1,1)
                    do kd = norm_id(1),norm_id(2)
                        res(:,:,kd-norm_id(1)+1,1) = &
                            &grid_eq%theta_F(:,:,kd)*eq_1%q_saf_E(kd,1) + &
                            &eq_2%L_E(:,:,kd,1,0,0)*eq_1%q_saf_E(kd,0)
                    end do
                    ! calculate T_EF(2,1)
                    do kd = norm_id(1),norm_id(2)
                        res(:,:,kd-norm_id(1)+1,2) = &
                            &(1._dp + eq_2%L_E(:,:,kd,0,1,0))*eq_1%q_saf_E(kd,0)
                    end do
                    ! calculate T_EF(3,1)
                    do kd = norm_id(1),norm_id(2)
                        res(:,:,kd-norm_id(1)+1,3) = -1._dp + &
                            &eq_2%L_E(:,:,kd,0,0,1)*eq_1%q_saf_E(kd,0)
                    end do
                    ! calculate T_EF(1,2)
                    do kd = norm_id(1),norm_id(2)
                        res(:,:,kd-norm_id(1)+1,4) = eq_1%flux_p_E(kd,1)/(2*pi)
                    end do
                    ! calculate T_EF(1,3)
                    res(:,:,:,7) = eq_2%L_E(:,:,norm_id(1):norm_id(2),1,0,0)
                    ! calculate T_EF(2,3)
                    res(:,:,:,8) = 1._dp + &
                        &eq_2%L_E(:,:,norm_id(1):norm_id(2),0,1,0)
                    ! calculate T_EF(3,3)
                    res(:,:,:,9) = eq_2%L_E(:,:,norm_id(1):norm_id(2),0,0,1)
                else                                                            ! using toroidal flux
                    ! calculate T_EF(1,1)
                    do kd = norm_id(1),norm_id(2)
                        res(:,:,kd-norm_id(1)+1,1) = - eq_2%L_E(:,:,kd,1,0,0) &
```

```
                        &+ grid_eq%zeta_E(:,:,kd)*eq_1%rot_t_E(kd,1)
                    end do
                    ! calculate T_EF(2,1)
                    res(:,:,:,2) = &
                        &- (1._dp + eq_2%L_E(:,:,norm_id(1):norm_id(2),0,1,0))
                    ! calculate T_EF(3,1)
                    do kd = norm_id(1),norm_id(2)
                        res(:,:,kd-norm_id(1)+1,3) = &
                            &eq_1%rot_t_E(kd,0) - eq_2%L_E(:,:,kd,0,0,1)
                    end do
                    ! calculate T_EF(1,2)
                    do kd = norm_id(1),norm_id(2)
                        res(:,:,kd-norm_id(1)+1,4) = &
                            &- eq_1%flux_t_E(kd,1)/(2*pi)
                    end do
                    ! calculate T_EF(3,3)
                    res(:,:,:,9) = -1._dp
                end if
            case (2)                                                    ! HELENA
                if (use_pol_flux_f) then                                ! using poloidal flux
                    ! calculate T_EF(1,1)
                    do kd = norm_id(1),norm_id(2)
                        res(:,:,kd-norm_id(1)+1,1) = &
                            &- eq_1%q_saf_E(kd,1)*grid_eq%theta_E(:,:,kd)
                    end do
                    ! calculate T_EF(2,1)
                    do kd = norm_id(1),norm_id(2)
                        res(:,:,kd-norm_id(1)+1,2) = - eq_1%q_saf_E(kd,0)
                    end do
                    ! calculate T_EF(3,1)
                    res(:,:,:,3) = 1._dp
                    ! calculate T_EF(1,2)
                    do kd = norm_id(1),norm_id(2)
                        res(:,:,kd-norm_id(1)+1,4) = eq_1%flux_p_E(kd,1)/(2*pi)
                    end do
                    ! calculate T_EF(2,3)
                    res(:,:,:,8) = 1._dp
                else                                                    ! using toroidal flux
                    ! calculate T_EF(1,1)
                    do kd = norm_id(1),norm_id(2)
                        res(:,:,kd-norm_id(1)+1,1) = &
                            &eq_1%rot_t_E(kd,1)*grid_eq%zeta_E(:,:,kd)
                    end do
                    ! calculate T_EF(2,1)
                    res(:,:,:,2) = -1._dp
                    ! calculate T_EF(3,1)
                    do kd = norm_id(1),norm_id(2)
                        res(:,:,kd-norm_id(1)+1,3) = eq_1%rot_t_E(kd,0)
                    end do
                    ! calculate T_EF(1,2)
                    do kd = norm_id(1),norm_id(2)
                        res(:,:,kd-norm_id(1)+1,4) = eq_1%flux_t_E(kd,1)/(2*pi)
                    end do
                    ! calculate T_EF(3,3)
                    res(:,:,:,9) = 1._dp
                end if
        end select

        ! set up plot variables for calculated values
        do id = 1,3
            do kd = 1,3
                ! user output
                call writo('Testing T_EF('//trim(i2str(kd))//','//&
                    &trim(i2str(id))//')')
                call lvl_ud(1)

                ! set some variables
                file_name = 'TEST_T_EF_'//trim(i2str(kd))//'_'//trim(i2str(id))
                description = 'Testing calculated with given value for T_EF('//&
                    &trim(i2str(kd))//','//trim(i2str(id))//')'

                ! plot difference
                call plot_diff_hdf5(res(:,:,:,c([kd,id],.false.)),&
                    &eq_2%T_EF(:,:,norm_id(1):norm_id(2),c([kd,id],.false.),&
                    &0,0,0),file_name,tot_dim,loc_offset,description,&
                    &output_message=.true.)

                call lvl_ud(-1)
            end do
        end do

        ! clean up
        call grid_trim%dealloc()

        ! user output
        call lvl_ud(-1)
        call writo('Test complete')
```

```
    end function test_t_ef

    !> Tests whether \f$ \frac{\partial^2}{\partial u_i \partial u_j} h_\text{H}
    !! \f$ is calculated correctly.
    !!
    !! \note Debug version only
    !!
    !! \return ierr
    integer function test_d12h_h(grid_eq,eq) result(ierr)
        use grid_utilities, only: trim_grid
        use num_utilities, only: c, spline
        use num_vars, only: norm_disc_prec_eq
        use helena_vars, only: ias

        character(*), parameter :: rout_name = 'test_D12h_H'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(eq_2_type), intent(in) :: eq                                       !< metric equilibrium

        ! local variables
        integer :: norm_id(2)                                                   ! untrimmed normal indices for trimmed grids
        integer :: id, jd, kd, ld                                               ! counters
        integer :: bcs(2,2)                                                     ! boundary conditions (theta(even), theta(odd))
        integer :: bc_ld(6)                                                     ! boundary condition type for each metric element
        real(dp) :: bcs_val(2,3)                                                ! values for boundary conditions
        real(dp), allocatable :: res(:,:,:,:)                                   ! result variable
        character(len=max_str_ln) :: file_name                                  ! name of plot file
        character(len=max_str_ln) :: description                                ! description of plot
        integer :: tot_dim(3), loc_offset(3)                                    ! total dimensions and local offset
        type(grid_type) :: grid_trim                                            ! trimmed equilibrium grid

        ! initialize ierr
        ierr = 0

        ! output
        call writo('Going to test whether D1 D2 h_H is calculated correctly')
        call lvl_ud(1)

        ! trim extended grid into plot grid
        ierr = trim_grid(grid_eq,grid_trim,norm_id)
        chckerr('')

        ! set up res
        allocate(res(grid_trim%n(1),grid_trim%n(2),grid_trim%loc_n_r,6))

        ! set total and local dimensions and local offset
        tot_dim = [grid_trim%n(1),grid_trim%n(2),grid_trim%n(3)]
        loc_offset = [0,0,grid_trim%i_min-1]

        ! set up boundary conditions
        if (ias.eq.0) then                                                      ! top-bottom symmetric
            bcs(:,1) = [1,1]                                                    ! theta(even): zero first derivative
            bcs(:,2) = [2,2]                                                    ! theta(odd): zero first derivative
        else
            bcs(:,1) = [-1,-1]                                                  ! theta(even): periodic
            bcs(:,2) = [-1,-1]                                                  ! theta(odd): periodic
        end if
        bcs_val = 0._dp

        ! boundary condition type for each metric element
        bc_ld = [1,2,0,1,0,1]                                                   ! 1: even, 2: odd, 0: zero

        ! calculate D1 D2 h_H alternatively
        do ld = 1,6
            do kd = norm_id(1),norm_id(2)
                do jd = 1,grid_trim%n(2)
                    if (bc_ld(ld).eq.0) cycle                                   ! quantity is zero

                    ierr = spline(grid_eq%theta_E(:,jd,kd),&
                        &eq%h_E(:,jd,kd,ld,0,1,0),grid_eq%theta_E(:,jd,kd),&
                        &res(:,jd,kd-norm_id(1)+1,ld),ord=norm_disc_prec_eq,&
                        &deriv=1,bcs=bcs(:,bc_ld(ld)),&
                        &bcs_val=bcs_val(:,bc_ld(ld)))
                    chckerr('')
                end do
            end do
        end do

        ! set up plot variables for calculated values
        do id = 1,3
            do kd = 1,3
                ! user output
                call writo('Testing h_H('//trim(i2str(kd))//','//&
                    &trim(i2str(id))//')')
```

```
                call lvl_ud(1)

                ! set some variables
                file_name = 'TEST_D12h_H_'//trim(i2str(kd))//'_'//&
                    &trim(i2str(id))
                description = 'Testing calculated with given value for D12h_H('&
                    &//trim(i2str(kd))//','//trim(i2str(id))//')'

                ! plot difference
                call plot_diff_hdf5(res(:,:,:,c([kd,id],.true.)),&
                    &eq%h_E(:,:,norm_id(1):norm_id(2),&
                    &c([kd,id],.true.),1,1,0),file_name,tot_dim,loc_offset,&
                    &description,output_message=.true.)

                call lvl_ud(-1)
            end do
        end do

        ! clean up
        call grid_trim%dealloc()

        ! user output
        call lvl_ud(-1)
        call writo('Test complete')
    end function test_d12h_h

    !> Performs  tests  on \f$  \mathcal{J}_\text{F}\f$.
    !!
    !!  - comparing it with the determinant of \f$g_\text{F}\f$
    !!  - comparing it with the direct formula
    !!
    !! \note Debug version only
    !!
    !! \return ierr
    integer function test_jac_f(grid_eq,eq_1,eq_2) result(ierr)
        use num_vars, only: eq_style, use_pol_flux_f
        use grid_utilities, only: trim_grid
        use num_utilities, only: calc_det
        use helena_vars, only: h_h_33, rbphi_h

        character(*), parameter :: rout_name = 'test_jac_F'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(eq_1_type), intent(in), target :: eq_1                             !< flux equilibrium
        type(eq_2_type), intent(in) :: eq_2                                     !< metric equilibrium

        ! local variables
        integer :: norm_id(2)                                                   ! untrimmed normal indices for trimmed grids
        real(dp), allocatable :: res(:,:,:)                                     ! result variable
        integer :: jd, kd                                                       ! counters
        character(len=max_str_ln) :: file_name                                  ! name of plot file
        character(len=max_str_ln) :: description                                ! description of plot
        integer :: tot_dim(3), loc_offset(3)                                    ! total dimensions and local offset
        type(grid_type) :: grid_trim                                            ! trimmed equilibrium grid
        real(dp), pointer :: dflux(:) => null()                                 ! points to D flux_p or D flux_t in E
coordinates
        integer :: pmone                                                        ! plus or minus one

        ! initialize ierr
        ierr = 0

        ! output
        call writo('Going to test the calculation of the Jacobian in Flux &
            &coordinates')
        call lvl_ud(1)

        ! trim extended grid into plot grid
        ierr = trim_grid(grid_eq,grid_trim,norm_id)
        chckerr('')

        ! set up res
        allocate(res(grid_trim%n(1),grid_trim%n(2),grid_trim%loc_n_r))

        ! set total and local dimensions and local offset
        tot_dim = [grid_trim%n(1),grid_trim%n(2),grid_trim%n(3)]
        loc_offset = [0,0,grid_trim%i_min-1]

        ! 1. Compare with determinant of g_F

        ! calculate Jacobian from determinant of g_F
        ierr = calc_det(res,eq_2%g_F(:,:,norm_id(1):norm_id(2),:,0,0,0),3)
        chckerr('')

        ! set some variables
        file_name = 'TEST_jac_F_1'
        description = 'Testing whether the Jacobian in Flux coordinates is &
```

```
            &consistent with determinant of metric matrix'

        ! plot difference
        call plot_diff_hdf5(res(:,:,:),&
            &eq_2%jac_F(:,:,norm_id(1):norm_id(2),0,0,0)**2,&
            &file_name,tot_dim,loc_offset,description,output_message=.true.)

        ! 2. Compare with explicit formula

        ! set up Dflux and pmone
        if (use_pol_flux_f) then                                        ! using poloidal flux
            dflux => eq_1%flux_p_E(:,1)
            pmone = 1                                                   ! flux_p_V = flux_p_F
        else
            dflux => eq_1%flux_t_E(:,1)
            pmone = -1                                                  ! flux_t_V = - flux_t_F
        end if

        ! calculate jac_F
        ! choose which equilibrium style is being used:
        !   1:  VMEC
        !   2:  HELENA
        select case (eq_style)
            case (1)                                                    ! VMEC
                do kd = norm_id(1),norm_id(2)
                    res(:,:,kd-norm_id(1)+1) = pmone * 2*pi * &
                        &eq_2%R_E(:,:,kd,0,0,0)*&
                        &(eq_2%R_E(:,:,kd,1,0,0)*eq_2%Z_E(:,:,kd,0,1,0) - &
                        &eq_2%Z_E(:,:,kd,1,0,0)*eq_2%R_E(:,:,kd,0,1,0)) / &
                        &(dflux(kd)*(1+eq_2%L_E(:,:,kd,0,1,0)))
                end do
            case (2)                                                    ! HELENA
                do kd = norm_id(1),norm_id(2)
                    do jd = 1,grid_trim%n(2)
                        res(:,jd,kd-norm_id(1)+1) = eq_1%q_saf_E(kd,0)/&
                            &(h_h_33(:,kd+grid_eq%i_min-1)*&
                            &rbphi_h(kd+grid_eq%i_min-1,0))             ! h_H_33 = 1/R^2 and RBphi_H = F are
    tabulated in eq. grid
                    end do
                end do
        end select

        ! set some variables
        file_name = 'TEST_jac_F_2'
        description = 'Testing whether the Jacobian in Flux coordinates is &
            &consistent with explicit formula'

        ! plot difference
        call plot_diff_hdf5(res(:,:,:),&
            &eq_2%jac_F(:,:,norm_id(1):norm_id(2),0,0,0),file_name,tot_dim,&
            &loc_offset,description,output_message=.true.)

        ! clean up
        nullify(dflux)
        call grid_trim%dealloc()

        ! user output
        call lvl_ud(-1)
        call writo('Test complete')
    end function test_jac_f

    !> Tests whether \f$g_\text{V}\f$ is calculated correctly.
    !!
    !! \note Debug version only
    !!
    !! \return ierr
    integer function test_g_v(grid_eq,eq) result(ierr)
        use grid_utilities, only: trim_grid
        use num_utilities, only: c

        character(*), parameter :: rout_name = 'test_g_V'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                          !< equilibrium grid
        type(eq_2_type), intent(in) :: eq                               !< metric equilibrium

        ! local variables
        integer :: norm_id(2)                                           ! untrimmed normal indices for trimmed grids
        integer :: id, kd                                               ! counters
        real(dp), allocatable :: res(:,:,:,:)                           ! result variable
        character(len=max_str_ln) :: file_name                          ! name of plot file
        character(len=max_str_ln) :: description                        ! description of plot
        integer :: tot_dim(3), loc_offset(3)                            ! total dimensions and local offset
        type(grid_type) :: grid_trim                                    ! trimmed equilibrium grid

        ! initialize ierr
        ierr = 0
```

```fortran

        ! output
        call writo('Going to test whether g_V is calculated correctly')
        call lvl_ud(1)

        ! trim extended grid into plot grid
        ierr = trim_grid(grid_eq,grid_trim,norm_id)
        chckerr('')

        ! set up res
        allocate(res(grid_trim%n(1),grid_trim%n(2),grid_trim%loc_n_r,6))

        ! set total and local dimensions and local offset
        tot_dim = [grid_trim%n(1),grid_trim%n(2),grid_trim%n(3)]
        loc_offset = [0,0,grid_trim%i_min-1]

        ! calculate g_V(1,1)
        res(:,:,:,1) = eq%R_E(:,:,norm_id(1):norm_id(2),1,0,0)**2 + &
            &eq%Z_E(:,:,norm_id(1):norm_id(2),1,0,0)**2
        ! calculate g_V(2,1)
        res(:,:,:,2) = eq%R_E(:,:,norm_id(1):norm_id(2),1,0,0)*&
            &eq%R_E(:,:,norm_id(1):norm_id(2),0,1,0) + &
            &eq%Z_E(:,:,norm_id(1):norm_id(2),1,0,0)*&
            &eq%Z_E(:,:,norm_id(1):norm_id(2),0,1,0)
        ! calculate g_V(3,1)
        res(:,:,:,3) = eq%R_E(:,:,norm_id(1):norm_id(2),1,0,0)*&
            &eq%R_E(:,:,norm_id(1):norm_id(2),0,0,1) + &
            &eq%Z_E(:,:,norm_id(1):norm_id(2),1,0,0)*&
            &eq%Z_E(:,:,norm_id(1):norm_id(2),0,0,1)
        ! calculate g_V(2,2)
        res(:,:,:,4) = eq%R_E(:,:,norm_id(1):norm_id(2),0,1,0)**2 + &
            &eq%Z_E(:,:,norm_id(1):norm_id(2),0,1,0)**2
        ! calculate g_V(3,2)
        res(:,:,:,5) = eq%R_E(:,:,norm_id(1):norm_id(2),0,1,0)*&
            &eq%R_E(:,:,norm_id(1):norm_id(2),0,0,1) + &
            &eq%Z_E(:,:,norm_id(1):norm_id(2),0,1,0)*&
            &eq%Z_E(:,:,norm_id(1):norm_id(2),0,0,1)
        ! calculate g_V(3,3)
        res(:,:,:,6) = eq%R_E(:,:,norm_id(1):norm_id(2),0,0,1)**2 + &
            &eq%Z_E(:,:,norm_id(1):norm_id(2),0,0,1)**2 + &
            &eq%R_E(:,:,norm_id(1):norm_id(2),0,0,0)**2

        ! set up plot variables for calculated values
        do id = 1,3
            do kd = 1,3
                ! user output
                call writo('Testing g_V('//trim(i2str(kd))//','//&
                    &trim(i2str(id))//')')
                call lvl_ud(1)

                ! set some variables
                file_name = 'TEST_g_V_'//trim(i2str(kd))//'_'//trim(i2str(id))
                description = 'Testing calculated with given value for g_V('//&
                    &trim(i2str(kd))//','//trim(i2str(id))//')'

                ! plot difference
                call plot_diff_hdf5(res(:,:,:,c([kd,id],.true.)),&
                    &eq%g_E(:,:,norm_id(1):norm_id(2),c([kd,id],.true.),&
                    &0,0,0),file_name,tot_dim,loc_offset,description,&
                    &output_message=.true.)

                call lvl_ud(-1)
            end do
        end do

        ! clean up
        call grid_trim%dealloc()

        ! user output
        call lvl_ud(-1)
        call writo('Test complete')
    end function test_g_v

    !> Tests whether \f$\mathcal{J}_\text{V}\f$ is calculated correctly.
    !!
    !! \note Debug version only
    !!
    !! \return ierr
    integer function test_jac_v(grid_eq,eq) result(ierr)
        use grid_utilities, only: trim_grid
        use num_utilities, only: calc_det

        character(*), parameter :: rout_name = 'test_jac_V'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(eq_2_type), intent(in) :: eq                                       !< metric equilibrium
```

```fortran

        ! local variables
        integer :: norm_id(2)                                           ! untrimmed normal indices for trimmed grids
        real(dp), allocatable :: res(:,:,:)                             ! result variable
        character(len=max_str_ln) :: file_name                          ! name of plot file
        character(len=max_str_ln) :: description                        ! description of plot
        integer :: tot_dim(3), loc_offset(3)                            ! total dimensions and local offset
        type(grid_type) :: grid_trim                                    ! trimmed equilibrium grid

        ! initialize ierr
        ierr = 0

        ! output
        call writo('Going to test the calculation of the Jacobian in the VMEC &
            &coords.')
        call lvl_ud(1)

        ! trim extended grid into plot grid
        ierr = trim_grid(grid_eq,grid_trim,norm_id)
        chckerr('')

        ! set up res
        allocate(res(grid_trim%n(1),grid_trim%n(2),grid_trim%loc_n_r))

        ! set total and local dimensions and local offset
        tot_dim = [grid_trim%n(1),grid_trim%n(2),grid_trim%n(3)]
        loc_offset = [0,0,grid_trim%i_min-1]

        ! calculate Jacobian from determinant of g_V
        ierr = calc_det(res,eq%g_E(:,:,norm_id(1):norm_id(2),:,0,0,0),3)
        chckerr('')

        ! set some variables
        file_name = 'TEST_jac_V'
        description = 'Testing whether the Jacobian in VMEC coordinates is &
            &consistent with determinant of metric matrix'

        ! plot difference
        call plot_diff_hdf5(-sqrt(res),&
            &eq%jac_E(:,:,norm_id(1):norm_id(2),0,0,0),&
            &file_name,tot_dim,loc_offset,description,output_message=.true.)

        ! clean up
        call grid_trim%dealloc()

        ! user output
        call lvl_ud(-1)
        call writo('Test complete')
    end function test_jac_v

    !> Tests whether \f$\vec{B}_\text{F}\f$ is calculated correctly.
    !!
    !! \note Debug version only
    !!
    !! \return ierr
    integer function test_b_f(grid_eq,eq_1,eq_2) result(ierr)
        use num_vars, only: eq_style
        use grid_utilities, only: trim_grid
        use num_utilities, only: c
        use vmec_utilities, only: fourier2real
        use vmec_vars, only: b_v_sub_s, b_v_sub_c, b_v_c, b_v_s, is_asym_v

        character(*), parameter :: rout_name = 'test_B_F'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                          !< equilibrium grid
        type(eq_1_type), intent(in) :: eq_1                             !< flux equilibrium
        type(eq_2_type), intent(in) :: eq_2                             !< metric equilibrium

        ! local variables
        integer :: norm_id(2)                                           ! untrimmed normal indices for trimmed grids
        integer :: norm_id_f(2)                                         ! norm_id transposed to full grid
        integer :: id, kd                                               ! counters
        real(dp), allocatable :: res(:,:,:,:)                           ! result variable
        real(dp), allocatable :: res2(:,:,:,:)                          ! result variable
        character(len=max_str_ln) :: file_name                          ! name of plot file
        character(len=max_str_ln) :: description                        ! description of plot
        integer :: tot_dim(3), loc_offset(3)                            ! total dimensions and local offset
        type(grid_type) :: grid_trim                                    ! trimmed equilibrium grid

        ! initialize ierr
        ierr = 0

        ! output
        call writo('Going to test whether the magnetic field is calculated &
            &correctly')
        call lvl_ud(1)
```

```fortran

        ! trim extended grid into plot grid
        ierr = trim_grid(grid_eq,grid_trim,norm_id)
        chckerr('')

        ! set norm_id_f for quantities tabulated on full grid
        norm_id_f = grid_eq%i_min+norm_id-1

        ! set up res and res2
        allocate(res(grid_trim%n(1),grid_trim%n(2),grid_trim%loc_n_r,4))
        allocate(res2(grid_trim%n(1),grid_trim%n(2),grid_trim%loc_n_r,4))

        ! set total and local dimensions and local offset
        tot_dim = [grid_trim%n(1),grid_trim%n(2),grid_trim%n(3)]
        loc_offset = [0,0,grid_trim%i_min-1]

        ! get covariant components and magnitude of B_E
        ! choose which equilibrium style is being used:
        !   1:  VMEC
        !   2:  HELENA
        select case (eq_style)
            case (1)                                                        ! VMEC
                do id = 1,3
                    ierr = fourier2real(&
                        &b_v_sub_c(:,norm_id_f(1):norm_id_f(2),id),&
                        &b_v_sub_s(:,norm_id_f(1):norm_id_f(2),id),&
                        &grid_eq%trigon_factors(:,:,:,norm_id(1):&
                        &norm_id(2),:),res(:,:,:,id))
                    chckerr('')
                end do
                ierr = fourier2real(b_v_c(:,norm_id_f(1):norm_id_f(2)),&
                    &b_v_s(:,norm_id_f(1):norm_id_f(2)),&
                    &grid_eq%trigon_factors(:,:,:,norm_id(1):norm_id(2),:),&
                    &res(:,:,:,4),[.true.,is_asym_v])
                chckerr('')
            case (2)                                                        ! HELENA
                res(:,:,:,1) = &
                    &eq_2%g_E(:,:,norm_id(1):norm_id(2),c([1,2],.true.),0,0,0)
                res(:,:,:,2) = &
                    &eq_2%g_E(:,:,norm_id(1):norm_id(2),c([2,2],.true.),0,0,0)
                do kd = norm_id(1),norm_id(2)
                    res(:,:,kd-norm_id(1)+1,3) = eq_1%q_saf_E(kd,0)*&
                        &eq_2%g_E(:,:,kd,c([3,3],.true.),0,0,0)
                    res(:,:,kd-norm_id(1)+1,4) = &
                        &sqrt(eq_2%g_E(:,:,kd,c([2,2],.true.),0,0,0) + &
                        &eq_1%q_saf_E(kd,0)**2*&
                        &eq_2%g_E(:,:,kd,c([3,3],.true.),0,0,0))
                end do
                do id = 1,4
                    do kd = norm_id(1),norm_id(2)
                        res(:,:,kd-norm_id(1)+1,id) = &
                            &res(:,:,kd-norm_id(1)+1,id)/&
                            &eq_2%jac_E(:,:,kd,0,0,0)
                    end do
                end do
        end select

        ! transform them to Flux coord. system
        res2 = 0._dp
        do id = 1,3
            do kd = 1,3
                res2(:,:,:,id) = res2(:,:,:,id) + &
                    &res(:,:,:,kd) * eq_2%T_FE(:,:,norm_id(1):&
                    &norm_id(2),c([id,kd],.false.),0,0,0)
            end do
        end do
        res2(:,:,:,4) = res(:,:,:,4)

        ! set up plot variables for calculated values
        do id = 1,4
            ! user output
            if (id.lt.4) then
                call writo('Testing B_F_'//trim(i2str(id)))
            else
                call writo('Testing B_F')
            end if
            call lvl_ud(1)

            ! set some variables
            if (id.lt.4) then
                file_name = 'TEST_B_F_'//trim(i2str(id))
                description = 'Testing calculated with given value for B_F_'//&
                    &trim(i2str(id))
            else
                file_name = 'TEST_B_F'
                description = 'Testing calculated with given value for B_F'
            end if
```

```fortran

            ! plot difference
            if (id.lt.4) then
                call plot_diff_hdf5(res2(:,:,:,id),&
                    &eq_2%g_F(:,:,norm_id(1):norm_id(2),c([3,id],.true.),0,0,0)&
                    &/eq_2%jac_F(:,:,norm_id(1):norm_id(2),0,0,0),file_name,&
                    &tot_dim,loc_offset,description,output_message=.true.)
            else
                call plot_diff_hdf5(res2(:,:,:,id),sqrt(&
                    &eq_2%g_F(:,:,norm_id(1):norm_id(2),c([3,3],.true.),0,0,0))&
                    &/eq_2%jac_F(:,:,norm_id(1):norm_id(2),0,0,0),file_name,&
                    &tot_dim,loc_offset,description,output_message=.true.)
            end if

            call lvl_ud(-1)
        end do

        ! clean up
        call grid_trim%dealloc()

        ! user output
        call lvl_ud(-1)
        call writo('Test complete')
    end function test_b_f

    !> Performs tests on pressure balance.
    !!
    !!  \f[\mu_0 \frac{\partial p}{\partial u^2} = \frac{1}{\mathcal{J}}
    !!      \left(\frac{\partial B_2}{\partial u^3} -
    !!      \frac{\partial B_3}{\partial u^2}\right)\f]
    !!  \f[\mu_0 \mathcal{J} \frac{\partial p}{\partial u^3} = 0 \rightarrow
    !!      \left(\frac{\partial B_1}{\partial u^3} =
    !!      \frac{\partial B_3}{\partial u^1}\right), \f]
    !! working in the (modified) Flux coordinates
    !!      \f$\left(\alpha,\psi,\theta\right)_\text{F}\f$
    !!
    !! \note Debug version only
    !!
    !! \return ierr
    integer function test_p(grid_eq,eq_1,eq_2) result(ierr)
        use num_utilities, only: c, spline
        use grid_utilities, only: trim_grid
        use eq_vars, only: vac_perm
        use num_vars, only: eq_style, norm_disc_prec_eq
        use helena_vars, only: rbphi_h, h_h_11, h_h_12

        character(*), parameter :: rout_name = 'test_p'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(eq_1_type), intent(in) :: eq_1                                     !< flux equilibrium variables
        type(eq_2_type), intent(in) :: eq_2                                     !< metric equilibrium variables

        ! local variables
        integer :: norm_id(2)                                                   ! untrimmed normal indices for trimmed grids
        integer :: norm_id_f(2)                                                 ! norm_id transposed to full grid
        real(dp), allocatable :: res(:,:,:,:)                                   ! result variable
        integer :: id, jd, kd                                                   ! counters
        character(len=max_str_ln) :: file_name                                  ! name of plot file
        character(len=max_str_ln) :: description                                ! description of plot
        integer :: tot_dim(3), loc_offset(3)                                    ! total dimensions and local offset
        type(grid_type) :: grid_trim                                            ! trimmed equilibrium grid

        ! initialize ierr
        ierr = 0

        ! output
        call writo('Going to test the consistency of the equilibrium variables &
            &with the given pressure')
        call lvl_ud(1)

        ! trim extended grid into plot grid
        ierr = trim_grid(grid_eq,grid_trim,norm_id)
        chckerr('')

        ! set up res
        allocate(res(grid_trim%n(1),grid_trim%n(2),grid_trim%loc_n_r,2))

        ! user output
        call writo('Checking if mu_0 D2 p = 1/J (D3 B_2 - D2_B3)')
        call lvl_ud(1)

        ! set total and local dimensions and local offset
        tot_dim = [grid_trim%n(1),grid_trim%n(2),grid_trim%n(3)]
        loc_offset = [0,0,grid_trim%i_min-1]

        ! calculate mu_0 D2p
```

```
        res(:,:,:,1) = &
            &(eq_2%g_FD(:,:,norm_id(1):norm_id(2),c([2,3],.true.),0,0,1) - &
            &eq_2%g_FD(:,:,norm_id(1):norm_id(2),c([3,3],.true.),0,1,0))/&
            &eq_2%jac_FD(:,:,norm_id(1):norm_id(2),0,0,0) - &
            &(eq_2%g_FD(:,:,norm_id(1):norm_id(2),c([2,3],.true.),0,0,0)*&
            &eq_2%jac_FD(:,:,norm_id(1):norm_id(2),0,0,1) - &
            &eq_2%g_FD(:,:,norm_id(1):norm_id(2),c([3,3],.true.),0,0,0)*&
            &eq_2%jac_FD(:,:,norm_id(1):norm_id(2),0,1,0))/ &
            &(eq_2%jac_FD(:,:,norm_id(1):norm_id(2),0,0,0)**2)
        res(:,:,:,1) = res(:,:,:,1)/eq_2%jac_FD(:,:,norm_id(1):norm_id(2),0,0,0)
        !!call plot_HDF5('var','TEST_Dg_FD_23',eq_2%g_FD(:,:,norm_id(1):norm_id(2),c([2,3],.true.),0,0,1))
        !!call plot_HDF5('var','TEST_g_FD_23',eq_2%g_FD(:,:,norm_id(1):norm_id(2),c([2,3],.true.),0,0,0))
        !!call plot_HDF5('var','TEST_Dg_FD_33',eq_2%g_FD(:,:,norm_id(1):norm_id(2),c([3,3],.true.),0,1,0))
        !!call plot_HDF5('var','TEST_g_FD_33',eq_2%g_FD(:,:,norm_id(1):norm_id(2),c([3,3],.true.),0,0,0))
        !!call plot_HDF5('var','TEST_Dg_FD',eq_2%jac_FD(:,:,norm_id(1):norm_id(2),0,0,1))
        !!call plot_HDF5('var','TEST_g_FD',eq_2%jac_FD(:,:,norm_id(1):norm_id(2),0,0,0))

        ! save mu_0 D2p in res
        do kd = norm_id(1),norm_id(2)
            res(:,:,kd-norm_id(1)+1,2) = vac_perm*eq_1%pres_FD(kd,1)
        end do

        ! set some variables
        file_name = 'TEST_D2p'
        description = 'Testing whether mu_0 D2 p = 1/J (D3 B_2 - D2_B3)'

        ! plot difference
        call plot_diff_hdf5(res(:,:,:,1),res(:,:,:,2),file_name,tot_dim,&
            &loc_offset,description,output_message=.true.)

        call lvl_ud(-1)

        ! user output
        call writo('Checking if mu_0 J D1p = 0 => D3 B_1 = D2 B_3')
        call lvl_ud(1)

        ! calculate D3 B1
        res(:,:,:,1) = &
            &eq_2%g_FD(:,:,norm_id(1):norm_id(2),c([1,3],.true.),0,0,1)/&
            &eq_2%jac_FD(:,:,norm_id(1):norm_id(2),0,0,0) - &
            &eq_2%g_FD(:,:,norm_id(1):norm_id(2),c([1,3],.true.),0,0,0)*&
            &eq_2%jac_FD(:,:,norm_id(1):norm_id(2),0,0,1) / &
            &eq_2%jac_FD(:,:,norm_id(1):norm_id(2),0,0,0)**2

        ! calculate D1 B3
        res(:,:,:,2) = &
            &eq_2%g_FD(:,:,norm_id(1):norm_id(2),c([3,3],.true.),1,0,0)/&
            &eq_2%jac_FD(:,:,norm_id(1):norm_id(2),0,0,0) - &
            &eq_2%g_FD(:,:,norm_id(1):norm_id(2),c([3,3],.true.),0,0,0)*&
            &eq_2%jac_FD(:,:,norm_id(1):norm_id(2),1,0,0) / &
            &eq_2%jac_FD(:,:,norm_id(1):norm_id(2),0,0,0)**2

        ! set some variables
        file_name = 'TEST_D1p'
        description = 'Testing whether D3 B_1 = D1 B_3'

        ! plot difference
        call plot_diff_hdf5(res(:,:,:,1),res(:,:,:,2),file_name,tot_dim,&
            &loc_offset,description,output_message=.true.)

        call lvl_ud(-1)

        ! extra testing if Helena
        if (eq_style.eq.2) then
            ! set norm_id_f for quantities tabulated on full grid
            norm_id_f = grid_eq%i_min+norm_id-1

            ! user output
            call writo('Checking if D2 B_3 |F = D1 (qF+qh_11/F) |H')
            call lvl_ud(1)

            ! calculate if D2 B_3 = D1 (qF) + D1 (q/F h_11)
            res(:,:,:,1) = &
                &(eq_2%g_FD(:,:,norm_id(1):norm_id(2),c([3,3],.true.),0,1,0)/&
                &eq_2%jac_FD(:,:,norm_id(1):norm_id(2),0,0,0) - &
                &eq_2%g_FD(:,:,norm_id(1):norm_id(2),c([3,3],.true.),0,0,0)*&
                &eq_2%jac_FD(:,:,norm_id(1):norm_id(2),0,1,0)/ &
                &(eq_2%jac_FD(:,:,norm_id(1):norm_id(2),0,0,0)**2))
            do jd = 1,grid_trim%n(2)
                do id = 1,grid_trim%n(1)
                    ierr = spline(grid_trim%loc_r_E,&
                        &eq_1%q_saf_E(norm_id(1):norm_id(2),0)*&
                        &rbphi_h(norm_id_f(1):norm_id_f(2),0)+&
                        &eq_1%q_saf_E(norm_id(1):norm_id(2),0)*&
                        &h_h_11(id,norm_id_f(1):norm_id_f(2))/&
                        &rbphi_h(norm_id_f(1):norm_id_f(2),0),&
                        &grid_trim%loc_r_E,res(id,jd,:,2),&
```

```
                        &ord=norm_disc_prec_eq,deriv=1)
                    chckerr('')
                end do
            end do

            ! set some variables
            file_name = 'TEST_D2B_3'
            description = 'Testing whether D2 B_3 |F = D1 (qF+qh_11/F) |H'

            ! plot difference
            call plot_diff_hdf5(res(:,:,:,1),res(:,:,:,2),file_name,tot_dim,&
                &loc_offset,description,output_message=.true.)

            call lvl_ud(-1)

            ! user output
            call writo('Checking if D3 B_2 |F = -q/F D2 h_11 + F q'' |H')
            call lvl_ud(1)

            ! calculate if D3 B_2 = -q/F D2 h_11 + F D1 q
            res(:,:,:,1) = &
                &eq_2%g_FD(:,:,norm_id(1):norm_id(2),c([2,3],.true.),0,0,1)/&
                &eq_2%jac_FD(:,:,norm_id(1):norm_id(2),0,0,0) - &
                &eq_2%g_FD(:,:,norm_id(1):norm_id(2),c([2,3],.true.),0,0,0)*&
                &eq_2%jac_FD(:,:,norm_id(1):norm_id(2),0,0,1)/ &
                &(eq_2%jac_FD(:,:,norm_id(1):norm_id(2),0,0,0)**2)
            do kd = norm_id(1),norm_id(2)
                do jd = 1,grid_trim%n(2)
                    ierr = spline(grid_eq%theta_E(:,jd,kd),&
                        &h_h_12(:,kd+grid_eq%i_min-1),grid_eq%theta_E(:,jd,kd),&
                        &res(:,jd,kd-norm_id(1)+1,2),&
                        &ord=norm_disc_prec_eq,deriv=1)
                    chckerr('')
                end do
                res(:,:,kd-norm_id(1)+1,2) = &
                    &-eq_1%q_saf_E(kd,0)/rbphi_h(kd+grid_eq%i_min-1,0) * &
                    &res(:,:,kd-norm_id(1)+1,2) + &
                    &rbphi_h(kd+grid_eq%i_min-1,0) * &
                    &eq_1%q_saf_E(kd,1)
            end do

            ! set some variables
            file_name = 'TEST_D3B_2'
            description = 'Testing whether D3 B_2 |F = -D2 (qh_12/F) + f q'' |H'

            ! plot difference
            call plot_diff_hdf5(res(:,:,:,1),res(:,:,:,2),file_name,tot_dim,&
                &loc_offset,description,output_message=.true.)

            call lvl_ud(-1)
        end if

        ! clean up
        call grid_trim%dealloc()

        ! user output
        call lvl_ud(-1)
        call writo('Test complete')
    end function test_p
#endif
end module eq_ops
```

D.20 /Users/toonweyens/Code/PB3D/Modules/eq_utilities.f90 File Reference

Interfaces and Types

- interface eq_utilities::calc_f_derivs

  Transforms derivatives of the equilibrium quantities in E coordinates to derivatives in the F coordinates.
- interface eq_utilities::calc_inv_met

  Calculate $D_1^{m_1} D_2^{m_2} D_3^{m_3} X$ *from* $D_1^{i_1} D_2^{i_2} D_3^{i_3} X$ *and* $D_1^{j_1} D_2^{j_2} D_3^{j_3} Y$ *where* $XY = 1$ *and* $i, j = 0 \dots m$*, according to [15].*
- interface eq_utilities::transf_deriv

  Calculates derivatives in a coordinate system B from derivatives in a coordinates system A, making use of the transformation matrix $\overline{\overline{T}}_B^A$.

Modules

- module eq_utilities

  Numerical utilities related to equilibrium variables.

Functions/Subroutines

- integer function, public eq_utilities::calc_g (g_a, t_ba, g_b, deriv, max_deriv)

  Calculate the metric coefficients in a coordinate system B ! using the.
- integer function, public eq_utilities::calc_memory_eq (arr_size, n_par, mem_size)

  Calculate memory in MB necessary for variables in equilibrium job.
- logical function, public eq_utilities::do_eq ()

  If this equilibrium job should be done, also increment `eq_job_nr`.
- elemental character(len=max_str_ln) function, public eq_utilities::eq_info ()

  Returns string with possible extension with equilibrium job as well as parallel job, or nothing if only one level and one parallel job.
- subroutine, public eq_utilities::print_info_eq (n_par_x_rich)

  Prints information for equilibrium parallel job.

Variables

- logical, public eq_utilities::debug_calc_inv_met_ind = .false.

  plot debug information for calc_inv_met_ind()

D.21 eq_utilities.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Numerical utilities related to equilibrium variables.
!-----------------------------------------------------------------------------!
module eq_utilities
#include <PB3D_macros.h>
    use str_utilities
    use output_ops
    use messages
    use num_vars, only: pi, dp, max_str_ln, max_deriv
    use grid_vars, only: grid_type
    use eq_vars, only: eq_1_type, eq_2_type
#if ldebug
    use num_utilities, only: check_deriv
#endif

    implicit none
    private
    public calc_inv_met, calc_g, transf_deriv, calc_f_derivs, do_eq, eq_info, &
        &print_info_eq, calc_memory_eq
#if ldebug
    public debug_calc_inv_met_ind
#endif

    ! global variables
#if ldebug
    !> \ldebug
    logical :: debug_calc_inv_met_ind = .false.                                !< plot debug information for calc_inv_met_ind()
#endif

    ! interfaces

    !> \public Transforms derivatives of the equilibrium quantities in E
    !! coordinates to derivatives in the F coordinates.
```

```
    !!
    !! \return ierr
    interface calc_f_derivs
        !> \public
        module procedure calc_f_derivs_1
        !> \public
        module procedure calc_f_derivs_2
    end interface

    !> \public Calculate
    !!      \f$D_1^{m_1} D_2^{m_2} D_3^{m_3} X\f$
    !! from
    !!      \f$D_1^{i_1} D_2^{i_2} D_3^{i_3} X\f$
    !! and
    !!      \f$D_1^{j_1} D_2^{j_2} D_3^{j_3} Y\f$
    !! where \f$XY=1\f$ and \f$i,j = 0\ldots m\f$, according to \cite Weyens3D.
    !!
    !!
    !! \f$D_1^{m_1} D_2^{m_2} D_3^{m_3} \f$ is defined as
    !!      \f$\left(\frac{\partial}{\partial u^1}\right)^{m_1}
    !!      \left(\frac{\partial}{\partial u^2}\right)^{m_2}
    !!      \left(\frac{\partial}{\partial u^3}\right)^{m_3} \f$
    !!
    !! \note It is assumed that the lower order derivatives have been calculated
    !! already. If not, the results will be incorrect.
    !!
    !! \return ierr
    interface calc_inv_met
        !> \public
        module procedure calc_inv_met_ind
        !> \public
        module procedure calc_inv_met_arr
        !> \public
        module procedure calc_inv_met_ind_0d
        !> \public
        module procedure calc_inv_met_arr_0d
    end interface

    !> \public Calculates derivatives in a  coordinate system B from derivatives
    !! in  a coordinates  system  A,  making use  of  the transformation  matrix
    !! \f$\overline{\text{T}}_\text{B}^\text{A}\f$.
    !!
    !! The routine works by exchanging the  derivatives in the coordinates B for
    !! derivatives in coordinates A using the formula
    !!  \f[\mathbf{D}_\text{B}^m X = \overline{\text{T}}_\text{B}^\text{A}
    !!      \mathbf{D}^1_\text{A} \left(\mathbf{D}^{m-1}_\text{B} X\right) , \f]
    !! where \f$\mathbf{D}^m\f$  is a tensor  of rank \f$m\f$ that  contains all
    !! the derivatives of total rank \f$m\f$. For example,
    !!  \f[\mathbf{D}^1 = \vec{D} =
    !!      \left(\begin{array}{c}\frac{\partial}{\partial u^1} \\
    !!      \frac{\partial}{\partial u^2} \\ \frac{\partial}{\partial u^3}
    !!      \end{array}\right)  \f]
    !!
    !! This  is done  for the  derivatives in  each of  the coordinates  B until
    !! degree 0 is reached.
    !!
    !! Furthermore, each of these degrees of derivatives in coordinates B can be
    !! be derived optionally  in the original coordinate system  A, which yields
    !! the formula:
    !!  \f[ \mathbf{D}^p_\text{A} \left(\mathbf{D}^m_\text{B} X \right) =
    !!      \sum_q \left(\begin{array}{c}p\\q\end{array}\right)
    !!      \mathbf{D}^q_\text{A}
    !!      \left(\overline{\text{T}}_\text{B}^\text{A}\right)
    !!      \left(\mathbf{D}^{p-q}_\text{A} \mathbf{D}^1_\text{A}\right)
    !!      \left( \mathbf{D}^{m-1}_\text{B} X \right)\f]
    !!
    !! This way, ultimately the desired derivatives  in the coordinates B can be
    !! obtained  recursively from  the lower  orders  in the  coordinates B  and
    !! higher orders in the coordinates A.
    !!
    !! For example:
    !!  - For order \f$M\f$, the formula can be used with \f$p=0\f$.
    !!  -   For   this,   it    is   necessary   that   \f$\mathbf{D}^1_\text{A}
    !!  \mathbf{D}^{m-1}_\text{B} X\f$ be precalculated, which can be done using
    !!  the formula with \f$p=1\f$ and \f$m=M-1\f$.
    !!  -   For   this,   it    is   necessary   that   \f$\mathbf{D}^1_\text{A}
    !!  \mathbf{D}^{M-1}_\text{B}     X\f$      and     \f$\mathbf{D}^2_\text{A}
    !!  \mathbf{D}^{M-1}_\text{B} X\f$ are precalculated, which can be done with
    !!  \f$p=1\f$ and \f$m=M-1\f$ as well as \f$p=2\f$ and \f$m=M-1\f$.
    !!  - etc.
    !!
    !! \see \cite Weyens3D for more detailed information.
    !!
    !! \return ierr
    interface transf_deriv
        !> \public
        module procedure transf_deriv_3_ind
```

```
        !> \public
        module procedure transf_deriv_3_arr
        !> \public
        module procedure transf_deriv_3_arr_2d
        !> \public
        module procedure transf_deriv_1_ind
    end interface

contains
    !> \private individual matrix version
    integer function calc_inv_met_ind(X,Y,deriv) result(ierr)
        use num_utilities, only: calc_inv, calc_mult, c, conv_mat
#if ldebug
        use num_vars, only: n_procs
#endif

        character(*), parameter :: rout_name = 'calc_inv_met_ind'

        ! input / output
        real(dp), intent(inout) :: x(1:,1:,1:,1:,0:,0:,0:)                    !< X
        real(dp), intent(in) :: y(1:,1:,1:,1:,0:,0:,0:)                       !< Y
        integer, intent(in) :: deriv(:)                                       !< derivatives

        ! local variables
        integer :: r, t, z                                                    ! counters for derivatives
        integer :: m1, m2, m3                                                 ! alias for deriv(i)
        integer :: dims(3)                                                    ! dimensions of coordinates
        real(dp) :: bin_fac(3)                                                ! binomial factor for norm., pol. and tor. sum
        real(dp), allocatable :: dum1(:,:,:,:)                                ! dummy variable representing metric matrix for some derivatives
        real(dp), allocatable :: dum2(:,:,:,:)                                ! dummy variable representing metric matrix for some derivatives
        integer :: nn                                                         ! size of matrices
        character(len=max_str_ln) :: err_msg                                  ! error message
#if ldebug
        real(dp), allocatable :: xy(:,:,:,:)                                  ! to check if XY is indeed unity
#endif

        ! initialize ierr
        ierr = 0

        ! set nn
        nn = size(x,4)

        ! tests
        if (size(y,4).ne.nn) then
            ierr = 1
            err_msg = 'Both matrices X and Y have to be either in full or &
                &lower diagonal storage'
            chckerr(err_msg)
        end if

        ! set mi
        m1 = deriv(1)
        m2 = deriv(2)
        m3 = deriv(3)

        ! set dims
        dims = [size(x,1),size(x,2),size(x,3)]

        ! initialize the requested derivative to zero
        x(:,:,:,:,m1,m2,m3) = 0._dp

        ! set up dummy variables
        allocate(dum1(dims(1),dims(2),dims(3),9))
        allocate(dum2(dims(1),dims(2),dims(3),9))

        ! initialize dum2
        dum2 = 0._dp

        ! calculate terms
        if (m1.eq.0 .and. m2.eq.0 .and. m3.eq.0) then                         ! direct inverse
            ierr = calc_inv(x(:,:,:,:,0,0,0),y(:,:,:,:,0,0,0),3)
            chckerr('')

#if ldebug
            ! check if X*Y is unity indeed if debugging
            if (debug_calc_inv_met_ind) then
                if (n_procs.gt.1) call writo('Debugging of calc_inv_met_ind &
                    &should be done with one processor only',warning=.true.)
                call writo('checking if X*Y = 1')
                call lvl_ud(1)
                allocate(xy(dims(1),dims(2),dims(3),9))
                ierr = calc_mult(x(:,:,:,:,0,0,0),y(:,:,:,:,0,0,0),xy,3)
                chckerr('')
                do r = 1,3
```

```
                    do t = 1,3
                        call writo(&
                            &trim(r2strt(minval(xy(:,1,:,c([r,t],.false.))))//&
                            &' < element ('//trim(i2str(t))//','//&
                            &trim(i2str(r))//') < '//&
                            &trim(r2strt(maxval(xy(:,1,:,c([r,t],.false.))))))
                    end do
                end do
                deallocate(xy)
                call lvl_ud(-1)
            end if
#endif
        else                                                            ! calculate using the other inverses
            do z = 0,m3                                                 ! derivs in third coord
                if (z.eq.0) then                                        ! first term in sum
                    bin_fac(3) = 1.0_dp
                else
                    bin_fac(3) = bin_fac(3)*(m3-(z-1))/z
                end if
                do t = 0,m2                                             ! derivs in second coord
                    if (t.eq.0) then                                    ! first term in sum
                        bin_fac(2) = 1.0_dp
                    else
                        bin_fac(2) = bin_fac(2)*(m2-(t-1))/t
                    end if
                    do r = 0,m1                                         ! derivs in first coord
                        if (r.eq.0) then                                ! first term in sum
                            bin_fac(1) = 1.0_dp
                        else
                            bin_fac(1) = bin_fac(1)*(m1-(r-1))/r
                        end if
                        if (z+t+r .lt. m1+m2+m3) then                   ! only add if not all r,t,z are equal to m1,m2,m3
                            ! multiply X(r,t,z) and Y(m1-r,m2-t,m3-z)
                            ierr = calc_mult(x(:,:,:,:,r,t,z),&
                                &y(:,:,:,:,m1-r,m2-t,m3-z),dum1,3)
                            chckerr('')
                            ! add result, multiplied by bin_fac, to dum2
                            dum2 = dum2 - product(bin_fac)*dum1
                        end if
                    end do
                end do
            end do

            ! right-multiply by X(0,0,0) and save in dum1
            ierr = calc_mult(dum2,x(:,:,:,:,0,0,0),dum1,3)
            chckerr('')
            ! save result in X(m1,m2,m3), converting if necessary
            ierr = conv_mat(dum1,x(:,:,:,:,m1,m2,m3),3)
            chckerr('')

            ! deallocate local variables
            deallocate(dum1,dum2)
        end if
    end function calc_inv_met_ind
    !> \private individual scalar version
    integer function calc_inv_met_ind_0d(X,Y,deriv) result(ierr)
        character(*), parameter :: rout_name = 'calc_inv_met_ind_0D'

        ! input / output
        real(dp), intent(inout) :: x(1:,1:,1:,0:,0:,0:)                 !< X
        real(dp), intent(in) :: y(1:,1:,1:,0:,0:,0:)                    !< Y
        integer, intent(in) :: deriv(:)                                 !< derivatives

        ! local variables
        integer :: r, t, z                                              ! counters for derivatives
        integer :: m1, m2, m3                                           ! alias for deriv(i)
        real(dp) :: bin_fac(3)                                          ! binomial factor for norm., pol. and tor. sum

        ! initialize ierr
        ierr = 0

        m1 = deriv(1)
        m2 = deriv(2)
        m3 = deriv(3)

        ! initialize the requested derivative
        x(:,:,:,m1,m2,m3) = 0.0_dp

        ! calculate terms
        if (m1.eq.0 .and. m2.eq.0 .and. m3.eq.0) then                   ! direct inverse
            x(:,:,:,0,0,0) = 1._dp/y(:,:,:,0,0,0)
        else                                                            ! calculate using the other inverses
            do z = 0,m3                                                 ! derivs in third coord
                if (z.eq.0) then                                        ! first term in sum
                    bin_fac(3) = 1.0_dp
```

```
                else
                    bin_fac(3) = bin_fac(3)*(m3-(z-1))/z
                    ! ADD SOME ERROR CHECKING HERE !!!!
                    chckerr('')
                end if
                do t = 0,m2                                                 ! derivs in second coord
                    if (t.eq.0) then                                        ! first term in sum
                        bin_fac(2) = 1.0_dp
                    else
                        bin_fac(2) = bin_fac(2)*(m2-(t-1))/t
                    end if
                    do r = 0,m1                                             ! derivs in first coord
                        if (r.eq.0) then                                    ! first term in sum
                            bin_fac(1) = 1.0_dp
                        else
                            bin_fac(1) = bin_fac(1)*(m1-(r-1))/r
                        end if
                        if (z+t+r .lt. m1+m2+m3) then                       ! only add if not all r,t,z are equal to m1,m2,m3
                            x(:,:,:,m1,m2,m3) = x(:,:,:,m1,m2,m3)-&
                                &bin_fac(1)*bin_fac(2)*bin_fac(3)* &
                                &x(:,:,:,r,t,z)*y(:,:,:,m1-r,m2-t,m3-z)
                        end if
                    end do
                end do
            end do

            ! right-multiply by X(0,0,0)
            x(:,:,:,m1,m2,m3) = x(:,:,:,m1,m2,m3)*x(:,:,:,0,0,0)
        end if
    end function calc_inv_met_ind_0d
    !< \private array matrix version
    integer function calc_inv_met_arr(X,Y,deriv) result(ierr)
        character(*), parameter :: rout_name = 'calc_inv_met_arr'

        ! input / output
        real(dp), intent(inout) :: x(1:,1:,1:,1:,0:,0:,0:)                  !< X
        real(dp), intent(in) :: y(1:,1:,1:,1:,0:,0:,0:)                     !< Y
        integer, intent(in) :: deriv(:,:)                                   !< derivatives

        ! local variables
        integer :: id

        ! initialize ierr
        ierr = 0

        do id = 1, size(deriv,2)
            ierr = calc_inv_met_ind(x,y,deriv(:,id))
            chckerr('')
        end do
    end function calc_inv_met_arr
    !< \private array scalar version
    integer function calc_inv_met_arr_0d(X,Y,deriv) result(ierr)
        character(*), parameter :: rout_name = 'calc_inv_met_arr_0D'

        ! input / output
        real(dp), intent(inout) :: x(1:,1:,1:,0:,0:,0:)                     !< X
        real(dp), intent(in) :: y(1:,1:,1:,0:,0:,0:)                        !< Y
        integer, intent(in) :: deriv(:,:)                                   !< derivatives

        ! local variables
        integer :: id

        ! initialize ierr
        ierr = 0

        do id = 1, size(deriv,2)
            ierr = calc_inv_met_ind_0d(x,y,deriv(:,id))
            chckerr('')
        end do
    end function calc_inv_met_arr_0d

    !> Calculate  the metric coefficients in  a coordinate system B  ! using the
    !metric  coefficients in  a coordinate  system  A and  the !  transformation
    !matrix \f$\overline{\text{T}}_\text{B}^\text{A}\f$.
    !!
    !! This is  based on the  general treatment  using the formula  described in
    !! \cite Weyens3D :
    !!  \f[ \vec{D}_1^{m_1} \vec{D}_2^{m_2} \vec{D}_3^{m_3} g_\text{B}
    !!      = \sum_1 \sum_2 \sum_3 C_1 C_2 C_3 \vec{D}^{i_1,j_1,k_1}
    !!      \overline{\text{T}}_\text{B}^\text{A}
    !!      \vec{D}^{i_2,j_2,k_2} g_\text{A} \vec{D}^{i_3,j_3,k_3}
    !!      \left(\overline{\text{T}}_\text{B}^\text{A}\right)^T \ ,
    !!  \f]
    !! with
    !!  \f[ k_l = m_l-i_l-j_l \ , \f]
    !! and with  the \f$\sum_i\f$  double summations, with  the first  one going
```

```
    !! from \f$0\f$ to \f$m_i\f$ and the second one from \f$m_j\f$ to \f$0\f$.
    !!
    !! The coefficients \f$C_l\f$ are calculated as
    !!  \f[
    !!      C_l = \left(\begin{array}{c}m_l\\i_l,j_l,k_l\end{array}\right)
    !!      \equiv \frac{m_l!}{i_l!j_l!(m_l-i_l-j_l)!} \ .
    !!  \f]
    !!
    !! \note It is assumed that the lower order derivatives have been calculated
    !! already. If not, the results will be incorrect. This is not checked.
    integer function calc_g(g_A,T_BA,g_B,deriv,max_deriv) result(ierr)
        use num_utilities, only: calc_mult, conv_mat

        character(*), parameter :: rout_name = 'calc_g'

        ! input / output
        real(dp), intent(in) :: g_a(:,:,:,:,0:,0:,0:)                          !< \f$g_\text{A}\f$
        real(dp), intent(in) :: t_ba(:,:,:,:,0:,0:,0:)                         !< \f$\overline{\text{T}}_\text{B}^\text{A}\f$
        real(dp), intent(inout) :: g_b(:,:,:,:,0:,0:,0:)                       !< \f$g_\text{B}\f$
        integer, intent(in) :: deriv(3)                                        !< derivatives
        integer, intent(in) :: max_deriv                                       !< maximum derivative

        ! local variables
        integer :: i1, j1, i2, j2, i3, j3, k1, k2, k3                          ! counters
        integer :: m1, m2, m3                                                  ! alias for deriv
        real(dp), allocatable :: c1(:), c2(:), c3(:)                           ! coeff. for (il)
        real(dp), allocatable :: dum1(:,:,:,:)                                 ! dummy variable representing metric matrix for some derivatives
        real(dp), allocatable :: dum2(:,:,:,:)                                 ! dummy variable representing metric matrix for some derivatives

        ! initialize ierr
        ierr = 0

#if ldebug
        ! check the derivatives requested
        ierr = check_deriv(deriv,max_deriv,'calc_g')
        chckerr('')
#endif

        ! set ml
        m1 = deriv(1)
        m2 = deriv(2)
        m3 = deriv(3)

        ! set up dummies
        allocate(dum1(size(g_a,1),size(g_a,2),size(g_a,3),9))
        allocate(dum2(size(g_a,1),size(g_a,2),size(g_a,3),9))

        ! initialize the requested derivative to zero
        g_b(:,:,:,:,m1,m2,m3) = 0.0_dp

        ! calculate the terms in the summation
        d1: do j1 = 0,m1                                                       ! derivatives in first coordinate
            call calc_c(m1,j1,c1)                                              ! calculate coeff. C1
            do i1 = m1-j1,0,-1
                d2: do j2 = 0,m2                                               ! derivatives in second coordinate
                    call calc_c(m2,j2,c2)                                      ! calculate coeff. C2
                    do i2 = m2-j2,0,-1
                        d3: do j3 = 0,m3                                       ! derivatives in third coordinate
                            call calc_c(m3,j3,c3)                              ! calculate coeff. C3
                            do i3 = m3-j3,0,-1
                                ! set up ki
                                k1 = m1 - j1 - i1
                                k2 = m2 - j2 - i2
                                k3 = m3 - j3 - i3
                                ! calculate term to add to the summation
                                ierr = calc_mult(g_a(:,:,:,:,j1,j2,j3),&
                                    &t_ba(:,:,:,:,k1,k2,k3),dum1,3,&
                                    &transp=[.false.,.true.])
                                chckerr('')
                                ierr = calc_mult(t_ba(:,:,:,:,i1,i2,i3),&
                                    &dum1,dum2,3)
                                chckerr('')
                                ! convert result to lower-diagonal storage
                                ierr = conv_mat(dum2,dum1(:,:,:,1:6),3)
                                chckerr('')
                                g_b(:,:,:,:,m1,m2,m3) = g_b(:,:,:,:,m1,m2,m3) &
                                    &+c1(i1)*c2(i2)*c3(i3)*dum1(:,:,:,1:6)
                            end do
                        end do d3
                    end do
                end do d2
            end do
        end do d1

```

```
        ! deallocate local variables
        deallocate(dum1,dum2)
    contains
        ! Calculate the coeff. C  at the all i values for  the current value for
        ! j, optionally making use of the coeff. C at previous value for j:
        ! - If it is the very first value (0,m), then it is just equal to 1
        ! - If  it is  the first value  in the current  i-summation, then  it is
        ! calculated from the first coeff. of the previous i-summation:
        !   C(j,m) = C(j-1,m) * (m-j+1)/j
        ! - If it is not the first  value in the current i-summation, then it is
        ! calculated from the previous value of the current i-summation
        !   C(j,i) = C(j,i+1) * (i+1)/(m-i-j)
        ! - If  it is  the last  value in  the current  i-summation, then  it is
        ! divided by 2 if (m-j) is even (see [ADD REF])
        !> \private
        subroutine calc_c(m,j,C)
            ! input / output
            integer, intent(in) :: m, j
            real(dp), intent(inout), allocatable :: c(:)

            ! local variables
            integer :: i                                                    ! counter
            real(dp) :: cm_prev

            ! if j>0, save the value Cm_prev
            if (j.gt.0) cm_prev = c(m-j+1)

            ! allocate C
            if (allocated(c)) deallocate (c)
            allocate(c(0:m-j))

            ! first value i = m-j
            if (j.eq.0) then                                                ! first coeff., for j = 0
                c(m) = 1.0_dp
            else                                                            ! first coeff., for j > 0 -> need value Cm_prev from previous array
                c(m-j) = (m-j+1.0_dp)/j * cm_prev
            end if

            ! all other values i = m-1 .. 0
            do i = m-j-1,0,-1
                c(i) = (i+1.0_dp)/(m-i-j) * c(i+1)
            end do
        end subroutine
    end function calc_g

    !> \private 3-D scalar version with one derivative
    integer recursive function transf_deriv_3_ind(x_a,t_ba,x_b,max_deriv,&
        &deriv_B,deriv_A_input) result(ierr)
        use num_utilities, only: add_arr_mult, c

        character(*), parameter :: rout_name = 'transf_deriv_3_ind'

        ! input / output
        real(dp), intent(in) :: x_a(1:,1:,1:,0:,0:,0:)                      !< variable and derivs. in coord. system A
        real(dp), intent(in) :: t_ba(1:,1:,1:,1:,0:,0:,0:)                  !< transf. mat. and derivs. between coord. systems A and B
        real(dp), intent(inout) :: x_b(1:,1:,1:)                            !< requested derivs. of variable in coord. system B
        integer, intent(in) :: max_deriv                                    !< maximum degrees of derivs.
        integer, intent(in) :: deriv_b(:)                                   !< derivs. in coord. system B
        integer, intent(in), optional :: deriv_a_input(:)                   !< derivs. in coord. system A (optional)

        ! local variables
        integer :: id, jd, kd, ld                                           ! counters
        integer :: deriv_id_b                                               ! holds the deriv. currently calculated
        integer, allocatable :: deriv_a(:)                                  ! holds either deriv_A or 0
        real(dp), allocatable :: x_b_x(:,:,:,:,:,:)                         ! X_B and derivs. D^p-q_A with extra 1 exchanged deriv. D_A
        integer, allocatable :: deriv_a_x(:)                                ! holds A derivs. for exchanged X_B_x
        integer, allocatable :: deriv_b_x(:)                                ! holds B derivs. for exchanged X_B_x

        ! initialize ierr
        ierr = 0

        ! setup deriv_A
        allocate(deriv_a(size(deriv_b)))
        if (present(deriv_a_input)) then
            deriv_a = deriv_a_input
        else
            deriv_a = 0
        end if

#if ldebug
        ! check the derivatives requested
        ! (every B deriv. needs all the A derivs. -> sum(deriv_B) needed)
        ierr = check_deriv(deriv_a + sum(deriv_b),max_deriv,'transf_deriv')
```

```
        chckerr('')
#endif

        ! detect first deriv. in the B coord. system that needs to be exchanged,
        ! with derivs in the A coord. system going from B coord. 1 to B coord. 2
        ! to B coord. 3
        ! if equal to  zero on termination, this means that  no more exchange is
        ! necessary
        deriv_id_b = 0
        do id = 1,3
            if (deriv_b(id).gt.0) then
                deriv_id_b = id
                exit
            end if
        end do

        ! calculate the  derivative in coord.  deriv_id_B of coord. system  B of
        ! one order lower than requested here
        ! check if we have reached the zeroth derivative in coord. B
        if (deriv_id_b.eq.0) then                                          ! return the function with its required A derivs.
            x_b = x_a(:,:,:,deriv_a(1),deriv_a(2),deriv_a(3))
        else                                                               ! apply the formula to obtain X_B using exchanged derivs.
            ! allocate  the  exchanged  X_B_x  and  the  derivs.  in  A  and  B,
            ! corresponding to D^m-1_B X
            allocate(x_b_x(size(x_b,1),size(x_b,2),size(x_b,3),&
                &0:deriv_a(1),0:deriv_a(2),0:deriv_a(3)))
            allocate(deriv_a_x(size(deriv_a)))
            allocate(deriv_b_x(size(deriv_b)))

            ! initialize X_B
            x_b = 0.0_dp

            ! loop over the 3 terms in the sum due to the transf. mat.
            do id = 1,3
                ! calculate X_B_x for orders 0 up to deriv_A
                do jd = 0,deriv_a(1)
                    do kd = 0,deriv_a(2)
                        do ld = 0,deriv_a(3)
                            ! calculate the  derivs. in  A for X_B_x:  for every
                            ! component in the sum due  to the transf. mat., the
                            ! deriv. is one order  higher than [jd,kd,ld] at the
                            ! corresponding coord. id
                            deriv_a_x = [jd,kd,ld]
                            deriv_a_x(id) = deriv_a_x(id) + 1

                            ! calculate the derivs. in B  for X_B_x: this is one
                            ! order lower in the coordinate deriv_id_B
                            deriv_b_x = deriv_b
                            deriv_b_x(deriv_id_b) = deriv_b_x(deriv_id_b) - 1

                            ! recursively call the subroutine again to calculate
                            ! X_B_x for the current derivatives
                            ierr = transf_deriv_3_ind(x_a,t_ba,&
                                &x_b_x(:,:,:,jd,kd,ld),max_deriv,deriv_b_x,&
                                &deriv_a_x)
                            chckerr('')
                        end do
                    end do
                end do

                ! use X_B_x at this term in summation due to the transf. mat. to
                ! update X_B using the formula
                ierr = add_arr_mult(&
                    &t_ba(:,:,:,c([deriv_id_b,id],.false.),0:,0:,0:),&
                    &x_b_x(:,:,:,0:,0:,0:),x_b,deriv_a)
                chckerr('')
            end do
        end if
    end function transf_deriv_3_ind
    !> \private 3-D scalar version with multiple derivatives
    integer function transf_deriv_3_arr(X_A,T_BA,X_B,max_deriv,derivs) &
        &result(ierr)
        character(*), parameter :: rout_name = 'transf_deriv_3_arr'

        ! input / output
        real(dp), intent(in) :: x_a(1:,1:,1:,0:,0:,0:)                     !< variable and derivs. in coord. system A
        real(dp), intent(in) :: t_ba(1:,1:,1:,1:,0:,0:,0:)                 !< transf. mat. and derivs. between coord. systems A and B
        real(dp), intent(inout) :: x_b(1:,1:,1:,0:,0:,0:)                  !< requested derivs. of variable in coord. system B
        integer, intent(in) :: max_deriv                                   !< maximum degrees of derivs.
        integer, intent(in) :: derivs(:,:)                                 !< series of derivs. (in coordinate system B)

        ! local variables
```

```
        integer :: id                                                   ! counter

        ! initialize ierr
        ierr = 0

        do id = 1, size(derivs,2)
            ierr = transf_deriv_3_ind(x_a,t_ba,&
                &x_b(:,:,:,derivs(1,id),derivs(2,id),derivs(3,id)),&
                &max_deriv,derivs(:,id))
            chckerr('')
        end do
    end function transf_deriv_3_arr
    !> \private 3-D matrix version  with multiple derivatives
    integer function transf_deriv_3_arr_2d(X_A,T_BA,X_B,max_deriv,derivs) &
        &result(ierr)
        use num_utilities, only: is_sym, c

        character(*), parameter :: rout_name = 'transf_deriv_3_arr_2D'

        ! input / output
        real(dp), intent(in) :: x_a(1:,1:,1:,1:,0:,0:,0:)              !< variable and derivs. in coord. system A
        real(dp), intent(in) :: t_ba(1:,1:,1:,1:,0:,0:,0:)             !< transf. mat. and derivs. between coord. systems A and B
        real(dp), intent(inout) :: x_b(1:,1:,1:,1:,0:,0:,0:)           !< requested derivs. of variable in coord. system B
        integer, intent(in) :: max_deriv                               !< maximum degrees of derivs.
        integer, intent(in) :: derivs(:,:)                             !< series of derivs. (in coordinate system B)

        ! local variables
        integer :: id, kd                                              ! counters
        character(len=max_str_ln) :: err_msg                           ! error message
        logical :: sym                                                 ! sym

        ! initialize ierr
        ierr = 0

        ! test whether the dimensions of X_A and X_B agree
        if (size(x_a,4).ne.size(x_b,4)) then
            err_msg = 'X_A and X_B need to have the same sizes'
            ierr = 1
            chckerr(err_msg)
        end if

        ierr = is_sym(3,size(x_a,4),sym)
        chckerr('')

        do kd = 1,size(x_a,4)
            do id = 1, size(derivs,2)
                ierr = transf_deriv_3_ind(x_a(:,:,:,kd,:,:,:),t_ba,&
                    &x_b(:,:,:,kd,derivs(1,id),derivs(2,id),derivs(3,id)),&
                    &max_deriv,derivs(:,id))
                chckerr('')
            end do
        end do
    end function transf_deriv_3_arr_2d
    !> \private 1-D scalar version with one derivative
    integer recursive function transf_deriv_1_ind(x_a,t_ba,x_b,max_deriv,&
        &deriv_B,deriv_A_input) result(ierr)
        use num_utilities, only: add_arr_mult

        character(*), parameter :: rout_name = 'transf_deriv_1_ind'

        ! input / output
        real(dp), intent(in) :: x_a(1:,0:)                             !< variable and derivs. in coord. system A
        real(dp), intent(inout) :: x_b(1:)                             !< requested derivs. of variable in coord. system B
        real(dp), intent(in) :: t_ba(1:,0:)                            !< transf. mat. and derivs. between coord. systems A and B
        integer, intent(in), optional :: deriv_a_input                 !< derivs. in coord. system A (optional)
        integer, intent(in) :: deriv_b                                 !< derivs. in coord. system B
        integer, intent(in) :: max_deriv                               !< maximum degrees of derivs.

        ! local variables
        integer :: jd                                                  ! counters
        integer :: deriv_a                                             ! holds either deriv_A or 0
        real(dp), allocatable :: x_b_x(:,:)                            ! X_B and derivs. D^p-q_A with extra 1 exchanged deriv. D_A
        integer :: deriv_a_x                                           ! holds A derivs. for exchanged X_B_x
        integer :: deriv_b_x                                           ! holds B derivs. for exchanged X_B_x

        ! initialize ierr
        ierr = 0

        ! setup deriv_A
        if (present(deriv_a_input)) then
            deriv_a = deriv_a_input
```

```
        else
            deriv_a = 0
        end if

#if ldebug
        ! check the derivatives requested
        ! (every B deriv. needs all the A derivs. -> sum(deriv_B) needed)
        ierr = check_deriv([deriv_a+deriv_b,0,0],max_deriv,'transf_deriv')
        chckerr('')
#endif

        ! calculate the  derivative in coord.  deriv_id_B of coord. system  B of
        ! one order lower than requested here
        ! check if we have reached the zeroth derivative in coord. B
        if (deriv_b.eq.0) then                                          ! return the function with its required A derivs.
            x_b = x_a(:,deriv_a)
        else                                                            ! apply the formula to obtain X_B using exchanged derivs.
            ! allocate  the  exchanged  X_B_x  and  the  derivs.  in  A  and  B,
            ! corresponding to D^m-1_B X
            allocate(x_b_x(size(x_b,1),0:deriv_a))

            ! initialize X_B
            x_b = 0.0_dp

            ! calculate X_B_x for orders 0 up to deriv_A
            do jd = 0,deriv_a
                ! calculate the derivs.  in A for X_B_x: for  every component in
                ! the  sum due  to the  transf. mat.,  the deriv.  is one  order
                ! higher than jd
                deriv_a_x = jd+1

                ! calculate the derivs. in B for X_B_x: this is one order lower
                deriv_b_x = deriv_b-1

                ! recursively call  the subroutine again to  calculate X_B_x for
                ! the current derivatives
                ierr = transf_deriv_1_ind(x_a,t_ba,x_b_x(:,jd),max_deriv,&
                    &deriv_b_x,deriv_a_input=deriv_a_x)
                chckerr('')
            end do

            ! use X_B_x to calculate X_B using the formula
            ierr = add_arr_mult(t_ba(:,0:),x_b_x(:,0:),x_b,[deriv_a,0,0])
            chckerr('')
        end if
    end function transf_deriv_1_ind

    !> \private flux version
    integer function calc_f_derivs_1(grid_eq,eq) result(ierr)
        use num_vars, only: eq_style, max_deriv, use_pol_flux_f
        use num_utilities, only: derivs, c, fac
        use eq_vars, only: max_flux_e

        character(*), parameter :: rout_name = 'calc_F_derivs_1'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                          !< equilibrium grid
        type(eq_1_type), intent(inout) :: eq                            !< flux equilibrium variables

        ! local variables
        integer :: id
        real(dp), allocatable :: t_fe_loc(:,:)                          ! T_FE(2,1)

        ! initialize ierr
        ierr = 0

        ! Transform depending on equilibrium style being used:
        !   1:  VMEC
        !   2:  HELENA
        select case (eq_style)
            case (1)                                                    ! VMEC
                ! user output
                call writo('Transform flux equilibrium quantities to flux &
                    &coordinates')

                call lvl_ud(1)

                ! set up local T_FE
                allocate(t_fe_loc(grid_eq%loc_n_r,0:max_deriv+1))
                if (use_pol_flux_f) then
                    do id = 0,max_deriv+1
                        t_fe_loc(:,id) = 2*pi/max_flux_e*eq%q_saf_E(:,id)
                    end do
                else
                    t_fe_loc(:,0) = -2*pi/max_flux_e
```

```
                    t_fe_loc(:,1:max_deriv+1) = 0._dp
                end if

                ! Transform the  derivatives in E coordinates  to derivatives in
                ! the F coordinates.
                do id = 0,max_deriv+1
                    ! pres_FD
                    ierr = transf_deriv(eq%pres_E,t_fe_loc,eq%pres_FD(:,id),&
                        &max_deriv+1,id)
                    chckerr('')

                    ! flux_p_FD
                    ierr = transf_deriv(eq%flux_p_E,t_fe_loc,&
                        &eq%flux_p_FD(:,id),max_deriv+1,id)
                    chckerr('')

                    ! flux_t_FD
                    ierr = transf_deriv(eq%flux_t_E,t_fe_loc,&
                        &eq%flux_t_FD(:,id),max_deriv+1,id)
                    chckerr('')
                    eq%flux_t_FD(:,id) = -eq%flux_t_FD(:,id)                    ! multiply by plus minus one

                    ! q_saf_FD
                    ierr = transf_deriv(eq%q_saf_E,t_fe_loc,eq%q_saf_FD(:,id),&
                        &max_deriv+1,id)
                    chckerr('')
                    eq%q_saf_FD(:,id) = -eq%q_saf_FD(:,id)                      ! multiply by plus minus one

                    ! rot_t_FD
                    ierr = transf_deriv(eq%rot_t_E,t_fe_loc,eq%rot_t_FD(:,id),&
                        &max_deriv+1,id)
                    chckerr('')
                    eq%rot_t_FD(:,id) = -eq%rot_t_FD(:,id)                      ! multiply by plus minus one
                end do

                call lvl_ud(-1)
            case (2)                                                            ! HELENA
                ! E derivatives are equal to F derivatives
                eq%flux_p_FD = eq%flux_p_E
                eq%flux_t_FD = eq%flux_t_E
                eq%pres_FD = eq%pres_E
                eq%q_saf_FD = eq%q_saf_E
                eq%rot_t_FD = eq%rot_t_E
        end select
    end function calc_f_derivs_1
    !> \private metric version
    integer function calc_f_derivs_2(eq) result(ierr)
        use num_vars, only: eq_style
        use num_utilities, only: derivs, c

        character(*), parameter :: rout_name = 'calc_F_derivs_2'

        ! input / output
        type(eq_2_type), intent(inout) :: eq                                    !< metric equilibrium variables

        ! local variables
        integer :: id
        integer :: pmone                                                        ! plus or minus one

        ! initialize ierr
        ierr = 0

        ! Set up pmone, depending on equilibrium style being used
        !   1:  VMEC
        !   2:  HELENA
        select case (eq_style)
            case (1)                                                            ! VMEC
                pmone = -1                                                      ! conversion VMEC LH -> RH coord. system
            case (2)                                                            ! HELENA
                pmone = 1
        end select

        ! user output
        call writo('Transform metric equilibrium quantities to flux &
            &coordinates')

        call lvl_ud(1)

        ! Transform the  derivatives in E coordinates  to derivatives in
        ! the F coordinates.
        do id = 0,max_deriv
            ! g_FD
            ierr = transf_deriv(eq%g_F,eq%T_FE,eq%g_FD,max_deriv,derivs(id))
            chckerr('')

            ! h_FD
            ierr = transf_deriv(eq%h_F,eq%T_FE,eq%h_FD,max_deriv,derivs(id))
```

```
            chckerr('')

            ! jac_FD
            ierr = transf_deriv(eq%jac_F,eq%T_FE,eq%jac_FD,max_deriv,&
                &derivs(id))
            chckerr('')
        end do

        call lvl_ud(-1)
    end function calc_f_derivs_2

    !> Calculate memory in MB necessary for variables in equilibrium job.
    !!
    !! The size of  these variables is equal to the  product of the non-parallel
    !! dimensions (e.g. \c n_geo x \c loc_n_r), times the number of variables.
    !!
    !! The latter should be:
    !!  -  PB3D:  only take  into  account  (\f$2\cdot6+1 =  13\f$)  equilibrium
    !!  variables \c g_FD, \c h_FD and  \c jac_FD, as the perturbation variables
    !!  are  divided in  jobs occupying  the remaning  space. These  equilibrium
    !!  variables are tabulated on the  equilibrium grid. Note that they contain
    !!  derivatives in extra dimensions, so that their size should be multiplied
    !!  by (\c max_deriv+1)^3.
    !!  - POST: take  into account these 13 equilibrium variables,  as well as 4
    !!  variables \c U and \c DU, with double size due to being complex, and the
    !!  additional dimension equal to \c n_mod_X, but without derivatives.
    !!
    !! \return ierr
    integer function calc_memory_eq(arr_size,n_par,mem_size) result(ierr)
        use iso_c_binding

        character(*), parameter :: rout_name = 'calc_memory_eq'

        ! input / output
        integer, intent(in) :: arr_size                                     !< size of part of X array
        integer, intent(in) :: n_par                                        !< number of parallel points
        real(dp), intent(inout) :: mem_size                                 !< total size

        ! local variables
        integer(C_SIZE_T) :: dp_size                                        ! size of dp

        ! initialize ierr
        ierr = 0

        call lvl_ud(1)

        ! get size of real variable
        dp_size = sizeof(1._dp)

        ! set memory size for metric equilibrium variables
        mem_size = 13._dp*arr_size
        mem_size = mem_size*n_par*(1+max_deriv)**3
        mem_size = mem_size*dp_size

        ! convert B to MB
        mem_size = mem_size*1.e-6_dp

        ! apply 50% safety factor (empirical)
        mem_size = mem_size*1.5_dp

        ! test overflow
        if (mem_size.lt.0) then
            ierr = 1
            chckerr('Overflow occured')
        end if

        call lvl_ud(-1)
    end function calc_memory_eq

    !> If this equilibrium job should be done, also increment \c eq_job_nr.
    logical function do_eq()
        use num_vars, only: eq_jobs_lims, eq_job_nr, rich_restart_lvl, &
            &jump_to_sol
        use rich_vars, only: rich_lvl

        ! increment equilibrium job nr.
        eq_job_nr = eq_job_nr + 1

        if (rich_lvl.eq.rich_restart_lvl .and. jump_to_sol) then            ! jumping to solution for restart level
            if (eq_job_nr.eq.1) then
                do_eq = .true.
            else
                do_eq = .false.
            end if
        else
            if (eq_job_nr.le.size(eq_jobs_lims,2)) then                     ! normal behavior, not yet at last
equilibrium job
```

```
                do_eq = .true.
            else                                                                    ! normal behavior, at last equilibrium job
                do_eq = .false.
            end if
        end if
    end function do_eq

    !> Returns string  with possible extension  with equilibrium job as  well as
    !! parallel job, or nothing if only one level and one parallel job.
    elemental character(len=max_str_ln) function eq_info()
        use num_vars, only: eq_jobs_lims, eq_job_nr

        if (size(eq_jobs_lims,2).gt.1) then
            eq_info = ' for Equilibrium job '//trim(i2str(eq_job_nr))//&
                &' of '//trim(i2str(size(eq_jobs_lims,2)))
        else
            eq_info = ''
        end if
    end function eq_info

    !> Prints information for equilibrium parallel job.
    subroutine print_info_eq(n_par_X_rich)
        use num_vars, only: eq_job_nr, eq_jobs_lims

        ! input / output
        integer, intent(in) :: n_par_x_rich                                         !< number of parallel points in this
    Richardson level

        ! user output
        if (size(eq_jobs_lims,2).gt.1) then
            call writo('but parallel limits for this job only ['//&
                &trim(i2str(eq_jobs_lims(1,eq_job_nr)))//'..'//&
                &trim(i2str(eq_jobs_lims(2,eq_job_nr)))//&
                &'] of [1..'//trim(i2str(n_par_x_rich))//']')
        end if
    end subroutine print_info_eq
end module eq_utilities
```

D.22 /Users/toonweyens/Code/PB3D/Modules/eq_vars.f90 File Reference

Interfaces and Types

- type eq_vars::eq_1_type

  flux equilibrium type
- type eq_vars::eq_2_type

  metric equilibrium type

Modules

- module eq_vars

  Variables that have to do with equilibrium quantities and the grid used in the calculations:

Functions/Subroutines

- subroutine eq_vars::init_eq_1 (eq, grid, setup_e, setup_f)

  Initializes new flux equilibrium.
- subroutine eq_vars::init_eq_2 (eq, grid, setup_e, setup_f)

  Initializes new metric equilibrium.
- subroutine eq_vars::copy_eq_1 (eq_i, grid_i, eq_o)

  Deep copy of flux equilibrium variables.
- subroutine eq_vars::copy_eq_2 (eq_i, grid_i, eq_o)

  Deep copy of metric equilibrium variables.
- subroutine eq_vars::dealloc_eq_1 (eq)

  Deallocates flux equilibrium quantities.
- subroutine eq_vars::dealloc_eq_2 (eq)

  Deallocates metric equilibrium quantities.

Variables

- real(dp), public eq_vars::r_0

  independent normalization constant for nondimensionalization
- real(dp), public eq_vars::pres_0

  independent normalization constant for nondimensionalization
- real(dp), public eq_vars::rho_0

  independent normalization constant for nondimensionalization
- real(dp), public eq_vars::b_0

  derived normalization constant for nondimensionalization
- real(dp), public eq_vars::psi_0

  derived normalization constant for nondimensionalization
- real(dp), public eq_vars::t_0

  derived normalization constant for nondimensionalization
- real(dp), public eq_vars::vac_perm = mu_0_original

  either usual mu_0 (default) or normalized
- real(dp), public eq_vars::max_flux_e

  max. flux in Equilibrium coordinates, set in calc_norm_range_PB3D_in
- real(dp), public eq_vars::max_flux_f

  max. flux in Flux coordinates, set in calc_norm_range_PB3D_in
- integer, public eq_vars::n_alloc_eq_1s

  nr. of allocated eq_1 variables
- integer, public eq_vars::n_alloc_eq_2s

  nr. of allocated eq_2 variables

D.23 eq_vars.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Variables that have to do with equilibrium quantities and the grid used in
!! the calculations:
!!  - The equilibrium variables are comprised  of the variables that result from
!!  the equilibrium calculation, such as pressure, rotational transform, etc.
!!  - The flux variables are tabulated on a 1D grid.
!!  - The metric variables are tabulated on a 3D grid
!!      + In the normal coordinate, they are tabulated in the equilibrium grid.
!!      + In  the angular coordinates, they  are tabulated in the  solution grid
!!      (VMEC), or  in the  equilibrium grid  followed by  an adaptation  to the
!!      solution grid (HELENA).
!!  - However, this  is not necessarily always  the case, as it  depends on the
!!  grid angles being  aligned with the grid (e.g. Providing  theta and zeta in
!!  the equilibrium grid so that the magnetic field lines are followed.
!!
!! \note In general in PB3D, there are two kinds of variables, differing from
!! one another in the way in which they are tabulated:
!!  - variables tabulated on the full output grid of the equilibrium code
!!  - variables tabulated in an internal grid of this code
!! \note In many places in the code a range in the normal coordinate is selected
!! for each  of the variables on  different processes. This selection  has to be
!! done correctly and  things can get a little bit  complicated if trimmed grids
!! are used (grids that have no overlap between processes).
!!
!! \see grid_ops()
!-----------------------------------------------------------------------------!
module eq_vars
#include <PB3D_macros.h>
    use str_utilities
    use messages
    use num_vars, only: dp, pi, max_str_ln, mu_0_original, weight_dp
    use grid_vars, only: grid_type

    implicit none
    private
```

```
    public r_0, pres_0, b_0, psi_0, rho_0, t_0, max_flux_e, max_flux_f, vac_perm
#if ldebug
    public n_alloc_eq_1s, n_alloc_eq_2s
#endif

    ! global variables
    real(dp) :: r_0                                                     !< independent normalization constant for nondimensionalization
    real(dp) :: pres_0                                                  !< independent normalization constant for nondimensionalization
    real(dp) :: rho_0                                                   !< independent normalization constant for nondimensionalization
    real(dp) :: b_0                                                     !< derived normalization constant for nondimensionalization
    real(dp) :: psi_0                                                   !< derived normalization constant for nondimensionalization
    real(dp) :: t_0                                                     !< derived normalization constant for nondimensionalization
    real(dp) :: vac_perm = mu_0_original                                !< either usual mu_0 (default) or normalized
    real(dp) :: max_flux_e                                              !< max. flux in Equilibrium coordinates, set in calc_norm_range_PB3D_in
    real(dp) :: max_flux_f                                              !< max. flux in Flux coordinates, set in calc_norm_range_PB3D_in
#if ldebug
    !> \ldebug
    integer :: n_alloc_eq_1s                                            !< nr. of allocated \c eq_1 variables
    !> \ldebug
    integer :: n_alloc_eq_2s                                            !< nr. of allocated \c eq_2 variables
#endif

    !> flux equilibrium type
    !!
    !! The arrays here are of the form
    !!  - <tt>(r)</tt>                        for variables without derivs.
    !!  - <tt>(r,Dr)</tt>                     for variables with derivs.
    type, public :: eq_1_type
        real(dp), allocatable :: pres_e(:,:)                            !< pressure, and norm. deriv.
        real(dp), allocatable :: q_saf_e(:,:)                           !< safety factor
        real(dp), allocatable :: rot_t_e(:,:)                           !< rot. transform
        real(dp), allocatable :: flux_p_e(:,:)                          !< poloidal flux and norm. deriv.
        real(dp), allocatable :: flux_t_e(:,:)                          !< toroidal flux and norm. deriv.
        real(dp), allocatable :: pres_fd(:,:)                           !< pressure, and norm. deriv.
        real(dp), allocatable :: q_saf_fd(:,:)                          !< safety factor
        real(dp), allocatable :: rot_t_fd(:,:)                          !< rot. transform
        real(dp), allocatable :: flux_p_fd(:,:)                         !< poloidal flux and norm. deriv.
        real(dp), allocatable :: flux_t_fd(:,:)                         !< toroidal flux and norm. deriv.
        real(dp), allocatable :: rho(:)                                 !< density
#if ldebug
        real(dp) :: estim_mem_usage(2)                                  !< expected memory usage \ldebug
#endif
    contains
        !> initialize
        procedure :: init => init_eq_1
        !> copy
        procedure :: copy => copy_eq_1
        !> deallocate
        procedure :: dealloc => dealloc_eq_1
    end type

    !> metric equilibrium type
    !!
    !! The arrays here are of the form
    !!  - <tt>(angle_1,angle_2,r)</tt>            for scalars without derivs.
    !!  - <tt>(angle_1,angle_2,r,D123)</tt>       for scalars with derivs.
    !!  - <tt>(angle_1,angle_2,r,6/9,D1,D2,D3)</tt> for tensors with derivs.
    !! \n  where it  is  refered to  the  discussion  of the  grid  type for  an
    !! explanation of the angles \c angle_1 and \c angle_2.
    !!
    !! The last index refers to the derivatives  in coordinate 1, 2 and 3, which
    !! refer to the coordinates described in \cite Weyens3D.
    !!  - For E(quilibrium) coordinates, they are
    !!      \f$\left(r,\theta,\zeta\right)_\text{E}\f$.
    !!  - For F(lux) coordinates, they are
    !!      \f$\left(\alpha,\psi,\gamma\right)_\text{F}\f$, where
    !!      + \f$\alpha = \zeta - q \theta\f$ and \f$\gamma = \theta\f$
    !!          for pol. flux,
    !!      + \f$\alpha = -\theta + \iota \zeta\f$ and \f$\gamma = \zeta\f$
    !!         Nfor tor. flux.
    !!
    !! The  fourth index  for  tensorial  variables correspond  to  the  9 or  6
    !! (symmetric) different values:
    !!  \f[ \left( \begin{array}{ccc}1&4&7\\2&5&8\\3&6&9\end{array} \right) \
    !!  \text{or} \quad
    !!  \left( \begin{array}{ccc}1& & \\2&4& \\3&5&6\end{array} \right) \f]
    !!
    !! \note Fortran only allows for 7 dimensions in arrays.
    type, public :: eq_2_type
```

```
        ! coordinate variables R, Z and L (VMEC)
        real(dp), allocatable :: r_e(:,:,:,:,:,:,:)                    !< R in E(quilibrium) coord
        real(dp), allocatable :: z_e(:,:,:,:,:,:,:)                    !< Z in E(quilibrium) coords.
        real(dp), allocatable :: l_e(:,:,:,:,:,:,:)                    !< L(ambda) in E(quilibrium) coords.
        ! upper (h) and lower (g) metric factors
        real(dp), allocatable :: g_c(:,:,:,:,:,:,:)                    !< in the C(ylindrical) coord. system
        real(dp), allocatable :: g_e(:,:,:,:,:,:,:)                    !< in the E(quilibrium) coord. system
        real(dp), allocatable :: h_e(:,:,:,:,:,:,:)                    !< in the E(quilibrium) coord. system
        real(dp), allocatable :: g_f(:,:,:,:,:,:,:)                    !< in the F(lux) coord. system with derivs. in V(MEC) system
        real(dp), allocatable :: h_f(:,:,:,:,:,:,:)                    !< in the F(lux) coord. system with derivs. in V(MEC) system
        real(dp), allocatable :: g_fd(:,:,:,:,:,:,:)                   !< in the F(lux) coord. system with derivs in F(lux) system
        real(dp), allocatable :: h_fd(:,:,:,:,:,:,:)                   !< in the F(lux) coord. system with derivs in F(lux) system
        ! transformation matrices
        real(dp), allocatable :: t_vc(:,:,:,:,:,:,:)                   !< C(ylindrical) to V(MEC)
        real(dp), allocatable :: t_ef(:,:,:,:,:,:,:)                   !< E(quilibrium) to F(lux)
        real(dp), allocatable :: t_fe(:,:,:,:,:,:,:)                   !< F(lux) to E(quilibrium)
        ! determinants of transformation matrices
        real(dp), allocatable :: det_t_ef(:,:,:,:,:,:)                 !< determinant of T_EF
        real(dp), allocatable :: det_t_fe(:,:,:,:,:,:)                 !< determinant of T_FE
        ! Jacobians
        real(dp), allocatable :: jac_e(:,:,:,:,:,:)                    !< jacobian of E(quilibrium) coord. system
        real(dp), allocatable :: jac_f(:,:,:,:,:,:)                    !< jacobian of F(lux) coord. system with derivs. in V(MEC) system
        real(dp), allocatable :: jac_fd(:,:,:,:,:,:)                   !< jacobian of F(lux) coord. system with derivs. in F(lux) system
        ! derived variables
        real(dp), allocatable :: s(:,:,:)                              !< magnetic shear
        real(dp), allocatable :: kappa_n(:,:,:)                        !< normal curvature
        real(dp), allocatable :: kappa_g(:,:,:)                        !< geodesic curvature
        real(dp), allocatable :: sigma(:,:,:)                          !< parallel current
#if ldebug
        real(dp) :: estim_mem_usage(2)                                 !< expected memory usage \ldebug
#endif
    contains
        !> initialize
        procedure :: init => init_eq_2
        !> copy
        procedure :: copy => copy_eq_2
        !> deallocate
        procedure :: dealloc => dealloc_eq_2
    end type

contains
    !> \public Initializes new flux equilibrium.
    !!
    !! The normal and angular grid can be in any coord. system, as only the grid
    !! sizes are used, not the coordinate values.
    !!
    !! Optionally, it can be chosen individually whether the E or F(D)
    !! quantities are allocated. D means that the derivatives are in F as well.
    !! The rationale behind this is that the E quantities are only used in the
    !! pre-perturbation phase.
    !! \note
    !! -# The E quantities are not written in eq_ops.print_output_eq().
    !! -# The quantities that do not have a derivative are considered F
    !! quantities. Alternatively, all quantities that have only one version,
    !! are considered F quantities, such as \c rho, \c kappa_n, ...
    !! -# The maximum derivative degree for flux quantities is one higher than
    !! \c max_deriv, because the derivative of some of them appear in the
    !! transformation matrices.
    subroutine init_eq_1(eq,grid,setup_E,setup_F)
        use num_vars, only: max_deriv
#if ldebug
        use num_vars, only: print_mem_usage, rank
#endif

        ! input / output
        class(eq_1_type), intent(inout) :: eq                          !< equilibrium to be initialized
        type(grid_type), intent(in) :: grid                            !< equilibrium grid
        logical, intent(in), optional :: setup_E                       !< whether to set up E
        logical, intent(in), optional :: setup_F                       !< whether to set up F

        ! local variables
        integer :: loc_n_r, n                                          ! local and total nr. of normal points
        logical :: setup_E_loc, setup_F_loc                            ! local versions of setup_E and setup_F
#if ldebug
        ! initialize memory usage
        if (print_mem_usage) eq%estim_mem_usage = 0._dp
#endif

        ! setup local setup_E and setup_F
        setup_e_loc = .true.
        if (present(setup_e)) setup_e_loc = setup_e
```

```
        setup_f_loc = .true.
        if (present(setup_f)) setup_f_loc = setup_f

        ! set local variables
        loc_n_r = grid%loc_n_r
        n = grid%n(3)

        if (setup_e_loc) then
            ! pres_E
            allocate(eq%pres_E(loc_n_r,0:max_deriv+1))

            ! q_saf_E
            allocate(eq%q_saf_E(loc_n_r,0:max_deriv+1))

            ! rot_t_E
            allocate(eq%rot_t_E(loc_n_r,0:max_deriv+1))

            ! flux_p_E
            allocate(eq%flux_p_E(loc_n_r,0:max_deriv+1))

            ! flux_t_E
            allocate(eq%flux_t_E(loc_n_r,0:max_deriv+1))

#if ldebug
            ! set estimated memory usage
            if (print_mem_usage) eq%estim_mem_usage(1) = &
                &eq%estim_mem_usage(1) + loc_n_r*(max_deriv+2)*5
#endif
        end if

        if (setup_f_loc) then
            ! pres_FD
            allocate(eq%pres_FD(loc_n_r,0:max_deriv+1))

            ! flux_p_FD
            allocate(eq%flux_p_FD(loc_n_r,0:max_deriv+1))

            ! flux_t_FD
            allocate(eq%flux_t_FD(loc_n_r,0:max_deriv+1))

            ! q_saf_FD
            allocate(eq%q_saf_FD(loc_n_r,0:max_deriv+1))

            ! rot_t_FD
            allocate(eq%rot_t_FD(loc_n_r,0:max_deriv+1))

            ! rho
            allocate(eq%rho(grid%loc_n_r))

#if ldebug
            ! set estimated memory usage
            if (print_mem_usage) eq%estim_mem_usage(2) = &
                &eq%estim_mem_usage(2) + loc_n_r*((max_deriv+2)*5+1)
#endif
        end if

#if ldebug
        ! increment n_alloc_eq_1s
        n_alloc_eq_1s = n_alloc_eq_1s + 1

        ! print memory usage
        if (print_mem_usage) call writo('[rank '//trim(i2str(rank))//&
            &' - Expected memory usage of eq_1: '//&
            &trim(r2strt(eq%estim_mem_usage(1)*0.008))//' kB for E and '//&
            &trim(r2strt(eq%estim_mem_usage(2)*0.008))//' kB for E]',&
            &alert=.true.)
#endif
    end subroutine init_eq_1

    !> \public Initializes new metric equilibrium.
    !!
    !! \see For explanation see init_eq_1().
    !!
    !! \note The  maximum derivative degree  for R, Z  and lambda is  one higher
    !! than \c max_deriv, because their  first derivative already appears in g_C
    !! and h_C.
    subroutine init_eq_2(eq,grid,setup_E,setup_F)                           ! metric version
        use num_vars, only: max_deriv, eq_style
#if ldebug
        use num_vars, only: print_mem_usage, rank
#endif

        ! input / output
        class(eq_2_type), intent(inout) :: eq                               !< equilibrium to be initialized
        type(grid_type), intent(in) :: grid                                 !< equilibrium grid
        logical, intent(in), optional :: setup_E                            !< whether to set up E
        logical, intent(in), optional :: setup_F                            !< whether to set up F
```

```

        ! local variables
        logical :: setup_E_loc, setup_F_loc                                     ! local versions of setup_E and setup_F
        integer :: dims(3)                                                      ! dimensions

        ! set up local variables
        dims = [grid%n(1),grid%n(2),grid%loc_n_r]

#if ldebug
        ! initialize memory usage
        if (print_mem_usage) eq%estim_mem_usage = 0._dp
#endif

        ! setup local setup_E and setup_F
        setup_e_loc = .true.
        if (present(setup_e)) setup_e_loc = setup_e
        setup_f_loc = .true.
        if (present(setup_f)) setup_f_loc = setup_f

        if (setup_e_loc) then
            ! initialize variables that are used for all equilibrium styles
            ! g_E
            allocate(eq%g_E(dims(1),dims(2),dims(3),6,&
                &0:max_deriv,0:max_deriv,0:max_deriv))

            ! h_E
            allocate(eq%h_E(dims(1),dims(2),dims(3),6,&
                &0:max_deriv,0:max_deriv,0:max_deriv))

            ! g_F
            allocate(eq%g_F(dims(1),dims(2),dims(3),6,&
                &0:max_deriv,0:max_deriv,0:max_deriv))

            ! h_F
            allocate(eq%h_F(dims(1),dims(2),dims(3),6,&
                &0:max_deriv,0:max_deriv,0:max_deriv))

            ! jac_E
            allocate(eq%jac_E(dims(1),dims(2),dims(3),&
                &0:max_deriv,0:max_deriv,0:max_deriv))

            ! jac_F
            allocate(eq%jac_F(dims(1),dims(2),dims(3),&
                &0:max_deriv,0:max_deriv,0:max_deriv))

            ! T_EF
            allocate(eq%T_EF(dims(1),dims(2),dims(3),9,&
                &0:max_deriv,0:max_deriv,0:max_deriv))

            ! T_FE
            allocate(eq%T_FE(dims(1),dims(2),dims(3),9,&
                &0:max_deriv,0:max_deriv,0:max_deriv))

            ! det_T_EF
            allocate(eq%det_T_EF(dims(1),dims(2),dims(3),&
                &0:max_deriv,0:max_deriv,0:max_deriv))

            ! det_T_FE
            allocate(eq%det_T_FE(dims(1),dims(2),dims(3),&
                &0:max_deriv,0:max_deriv,0:max_deriv))

#if ldebug
            ! set estimated memory usage
            if (print_mem_usage) eq%estim_mem_usage(1) = &
                &eq%estim_mem_usage(1) + product(dims)*&
                &((max_deriv+1)**3*(4*6+2*9+4))
#endif

            ! initialize  variables that  are  specificic  to which  equilibrium
            ! style is being used:
            !   1:  VMEC
            !   2:  HELENA
            select case (eq_style)
                case (1)                                                        ! VMEC
                    ! g_C
                    allocate(eq%g_C(dims(1),dims(2),dims(3),6,&
                        &0:max_deriv,0:max_deriv,0:max_deriv))

                    ! T_VC
                    allocate(eq%T_VC(dims(1),dims(2),dims(3),9,&
                        &0:max_deriv,0:max_deriv,0:max_deriv))

                    ! R
                    allocate(eq%R_E(dims(1),dims(2),dims(3),&
                        &0:max_deriv+1,0:max_deriv+1,0:max_deriv+1))

                    ! Z
```

```
                    allocate(eq%Z_E(dims(1),dims(2),dims(3),&
                        &0:max_deriv+1,0:max_deriv+1,0:max_deriv+1))

                    ! lambda
                    allocate(eq%L_E(dims(1),dims(2),dims(3),&
                        &0:max_deriv+1,0:max_deriv+1,0:max_deriv+1))

#if ldebug
                    ! set estimated memory usage
                    if (print_mem_usage) eq%estim_mem_usage(1) = &
                        &eq%estim_mem_usage(1) + product(dims)*(&
                        &(max_deriv+1)**3*(1*6+1*9+2) + &
                        &(max_deriv+2)**3*(3))
#endif
                case (2)                                                    ! HELENA
                    ! do nothing
            end select
        end if

        if (setup_f_loc) then
            ! initialize variables that are used for all equilibrium styles
            ! g_FD
            allocate(eq%g_FD(dims(1),dims(2),dims(3),6,&
                &0:max_deriv,0:max_deriv,0:max_deriv))

            ! h_FD
            allocate(eq%h_FD(dims(1),dims(2),dims(3),6,&
                &0:max_deriv,0:max_deriv,0:max_deriv))

            ! jac_FD
            allocate(eq%jac_FD(dims(1),dims(2),dims(3),&
                &0:max_deriv,0:max_deriv,0:max_deriv))

            ! magnetic shear
            allocate(eq%S(dims(1),dims(2),dims(3)))

            ! normal curvature
            allocate(eq%kappa_n(dims(1),dims(2),dims(3)))

            ! geodesic curvature
            allocate(eq%kappa_g(dims(1),dims(2),dims(3)))

            ! parallel current
            allocate(eq%sigma(dims(1),dims(2),dims(3)))

#if ldebug
            ! set estimated memory usage
            if (print_mem_usage) eq%estim_mem_usage(2) = &
                &eq%estim_mem_usage(2) + product(dims)*(&
                &(max_deriv+1)**3*(2*6+1) + &
                &4)
#endif
        end if

#if ldebug
        ! increment n_alloc_eq_2s
        n_alloc_eq_2s = n_alloc_eq_2s + 1

        ! print memory usage
        if (print_mem_usage) call writo('[rank '//trim(i2str(rank))//&
            &' - Expected memory usage of eq_2: '//&
            &trim(r2strt(eq%estim_mem_usage(1)*0.008))//' kB for E and '//&
            &trim(r2strt(eq%estim_mem_usage(2)*0.008))//' kB for E]',&
            &alert=.true.)
#endif
    end subroutine init_eq_2

    !> \public Deep copy of flux equilibrium variables.
    subroutine copy_eq_1(eq_i,grid_i,eq_o)
        use grid_vars, only: grid_type

        ! input / output
        class(eq_1_type), intent(in) :: eq_i                                !< eq_1 to be copied
        type(grid_type), intent(in) :: grid_i                               !< grid of eq_i
        type(eq_1_type), intent(inout) :: eq_o                              !< copied eq_1

        ! local variables
        logical :: setup_E
        logical :: setup_F

        setup_e = allocated(eq_i%pres_E)
        setup_f = allocated(eq_i%pres_FD)

        call eq_o%init(grid_i,setup_e=setup_e,setup_f=setup_f)

        if (setup_e) then
            if (allocated(eq_i%pres_E)) eq_o%pres_E = eq_i%pres_E
```

```
            if (allocated(eq_i%q_saf_E)) eq_o%q_saf_E = eq_i%q_saf_E
            if (allocated(eq_i%rot_t_E)) eq_o%rot_t_E = eq_i%rot_t_E
            if (allocated(eq_i%flux_p_E)) eq_o%flux_p_E = eq_i%flux_p_E
            if (allocated(eq_i%flux_t_E)) eq_o%flux_t_E = eq_i%flux_t_E
        end if

        if (setup_f) then
            if (allocated(eq_i%pres_FD)) eq_o%pres_FD = eq_i%pres_FD
            if (allocated(eq_i%flux_p_FD)) eq_o%flux_p_FD = eq_i%flux_p_FD
            if (allocated(eq_i%flux_t_FD)) eq_o%flux_t_FD = eq_i%flux_t_FD
            if (allocated(eq_i%q_saf_FD)) eq_o%q_saf_FD = eq_i%q_saf_FD
            if (allocated(eq_i%rot_t_FD)) eq_o%rot_t_FD = eq_i%rot_t_FD
            if (allocated(eq_i%rho)) eq_o%rho = eq_i%rho
        end if
    end subroutine copy_eq_1

    !> \public Deep copy of metric equilibrium variables.
    !!
    !! \return ierr
    subroutine copy_eq_2(eq_i,grid_i,eq_o)
        use num_vars, only: eq_style
        use grid_vars, only: grid_type

        ! input / output
        class(eq_2_type), intent(in) :: eq_i                                    !< eq_2 to be copied
        type(grid_type), intent(in) :: grid_i                                   !< grid of eq_i
        type(eq_2_type), intent(inout) :: eq_o                                  !< copied eq_1

        ! local variables
        logical :: setup_E
        logical :: setup_F

        setup_e = allocated(eq_i%jac_E)
        setup_f = allocated(eq_i%jac_FD)

        call eq_o%init(grid_i,setup_e=setup_e,setup_f=setup_f)

        if (setup_e) then
            if (allocated(eq_i%g_E)) eq_o%g_E = eq_i%g_E
            if (allocated(eq_i%h_E)) eq_o%h_E = eq_i%h_E
            if (allocated(eq_i%g_F)) eq_o%g_F = eq_i%g_F
            if (allocated(eq_i%h_F)) eq_o%h_F = eq_i%h_F
            if (allocated(eq_i%jac_E)) eq_o%jac_E = eq_i%jac_E
            if (allocated(eq_i%jac_F)) eq_o%jac_F = eq_i%jac_F
            if (allocated(eq_i%T_EF)) eq_o%T_EF = eq_i%T_EF
            if (allocated(eq_i%T_FE)) eq_o%T_FE = eq_i%T_FE
            if (allocated(eq_i%det_T_EF)) eq_o%det_T_EF = eq_i%det_T_EF
            if (allocated(eq_i%det_T_FE)) eq_o%det_T_FE = eq_i%det_T_FE

            select case (eq_style)
                case (1)                                                        ! VMEC
                    if (allocated(eq_i%g_C)) eq_o%g_C = eq_i%g_C
                    if (allocated(eq_i%T_VC)) eq_o%T_VC = eq_i%T_VC
                    if (allocated(eq_i%R_E)) eq_o%R_E = eq_i%R_E
                    if (allocated(eq_i%Z_E)) eq_o%Z_E = eq_i%Z_E
                    if (allocated(eq_i%l_E)) eq_o%l_E = eq_i%l_E
                case (2)                                                        ! HELENA
                    ! do nothing
            end select
        end if

        if (setup_f) then
            if (allocated(eq_i%g_FD)) eq_o%g_FD = eq_i%g_FD
            if (allocated(eq_i%h_FD)) eq_o%h_FD = eq_i%h_FD
            if (allocated(eq_i%jac_FD)) eq_o%jac_FD = eq_i%jac_FD
            if (allocated(eq_i%S)) eq_o%S = eq_i%S
            if (allocated(eq_i%kappa_n)) eq_o%kappa_n = eq_i%kappa_n
            if (allocated(eq_i%kappa_g)) eq_o%kappa_g = eq_i%kappa_g
            if (allocated(eq_i%sigma)) eq_o%sigma = eq_i%sigma
        end if
    end subroutine copy_eq_2

    !> \public Deallocates flux equilibrium quantities.
    subroutine dealloc_eq_1(eq)
#if ldebug
        use num_vars, only: rank, print_mem_usage
#endif

        ! input / output
        class(eq_1_type), intent(inout) :: eq                                   !< equilibrium to be deallocated

#if ldebug
        ! local variables
        integer :: mem_diff                                                     ! difference in memory
        real(dp) :: estim_mem_usage                                             ! estimated memory usage

        ! memory usage before deallocation
```

```
        if (print_mem_usage) then
            mem_diff = get_mem_usage()
            estim_mem_usage = sum(eq%estim_mem_usage)
        end if
#endif

        ! deallocate allocatable variables
        call dealloc_eq_1_final(eq)

#if ldebug
        ! decrement n_alloc_eq_1s
        n_alloc_eq_1s = n_alloc_eq_1s - 1

        ! memory usage difference after deallocation
        if (print_mem_usage) then
            mem_diff = mem_diff - get_mem_usage()
            call writo('[Rank '//trim(i2str(rank))//' - liberated '//&
                &trim(i2str(mem_diff))//'kB deallocating eq_1 ('//&
                &trim(i2str(nint(100*mem_diff/&
                &(estim_mem_usage*weight_dp))))//&
                &'% of estimated)]',alert=.true.)
        end if
#endif
    contains
        ! Note: intent(out) automatically deallocates the variable
        !> \private
        subroutine dealloc_eq_1_final(eq)
            ! input / output
            type(eq_1_type), intent(out) :: eq                                  ! equilibrium to be deallocated
        end subroutine dealloc_eq_1_final
    end subroutine dealloc_eq_1

    !> \public Deallocates metric equilibrium quantities.
    subroutine dealloc_eq_2(eq)
#if ldebug
        use num_vars, only: rank, print_mem_usage
#endif
        ! input / output
        class(eq_2_type), intent(inout) :: eq                                   !< equilibrium to be deallocated

#if ldebug
        ! local variables
        integer :: mem_diff                                                     ! difference in memory
        real(dp) :: estim_mem_usage                                             ! estimated memory usage

        ! memory usage before deallocation
        if (print_mem_usage) then
            mem_diff = get_mem_usage()
            estim_mem_usage = sum(eq%estim_mem_usage)
        end if
#endif

        ! deallocate allocatable variables
        call dealloc_eq_2_final(eq)

#if ldebug
        ! decrement n_alloc_eq_2s
        n_alloc_eq_2s = n_alloc_eq_2s - 1

        ! memory usage difference after deallocation
        if (print_mem_usage) then
            mem_diff = mem_diff - get_mem_usage()
            call writo('[Rank '//trim(i2str(rank))//' - liberated '//&
                &trim(i2str(mem_diff))//'kB deallocating eq_2 ('//&
                &trim(i2str(nint(100*mem_diff/&
                &(estim_mem_usage*weight_dp))))//&
                &'% of estimated)]',alert=.true.)
        end if
#endif
    contains
        ! Note: intent(out) automatically deallocates the variable
        !> \private
        subroutine dealloc_eq_2_final(eq)
            ! input / output
            type(eq_2_type), intent(out) :: eq                                  ! equilibrium to be deallocated
        end subroutine dealloc_eq_2_final
    end subroutine dealloc_eq_2
end module eq_vars
```

D.24 /Users/toonweyens/Code/PB3D/Modules/files_ops.f90 File Reference

Modules

- module files_ops

  Operations related to files !

Functions/Subroutines

- subroutine, public files_ops::init_files ()

  Initialize the variables for the module.
- integer function, public files_ops::parse_args ()

  Parses the command line arguments.
- integer function, public files_ops::open_input ()

  Open the input files.
- integer function, public files_ops::open_output ()

  Open the output files.
- subroutine, public files_ops::close_output ()

  Closes the output file.

Variables

- character(len=max_str_ln), dimension(:), allocatable, public files_ops::opt_args

  optional arguments that can be passed using `--[name]`

D.25 files_ops.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------!
!> Operations related to files                                          !
!-----------------------------------------------------------------------!
module files_ops
#include <PB3D_macros.h>
    use str_utilities
    use messages
    use num_vars, only: dp, max_str_ln
    implicit none
    private
    public open_input, open_output, parse_args, init_files, close_output, &
        &opt_args

    ! user-specified arguments
    integer :: numargs                                          ! control the user-specified arguments
    character(len=max_str_ln), allocatable :: command_arg(:)    ! passeed command-line arguments

    ! options provided with command line
    character(len=max_str_ln), allocatable :: opt_args(:)       !< optional arguments that can be passed using <tt>--[name]</tt>
    integer, allocatable :: inc_args(:)                         !< number of arguments for each optional argument

contains
    !> Initialize the variables for the module.
    subroutine init_files()
        use num_vars, only: ltest, prog_style

```

```fortran
        ! select according to program style
        select case (prog_style)
            case(1)                                                         ! PB3D
                allocate(opt_args(25), inc_args(25))
                opt_args = ''
                inc_args = 0
                opt_args(7) = '--no_guess'
                opt_args(8) = '--jump_to_sol'
                opt_args(9) = '--export_HEL'
                opt_args(10) = '--plot_VMEC_modes'
                opt_args(11) = '--invert_top_bottom_H'
                opt_args(12) = '-st_pc_factor_shift_type'
                opt_args(13) = '-st_pc_type'
                opt_args(14) = '-st_pc_factor_mat_solver_package'
                opt_args(15) = '-eps_type'
                opt_args(16) = '-eps_monitor'
                opt_args(17) = '-eps_tol'
                opt_args(18) = '-eps_ncv'
                opt_args(19) = '-eps_mpd'
                opt_args(20) = '-eps_view'
                opt_args(21) = '-st_type'
                opt_args(22) = '-st_pc_type'
                opt_args(23) = '-st_pc_factor_mat_solver_package'
                opt_args(24) = '-log_view'
                opt_args(25) = '-st_ksp_type'
                inc_args(7:25) = [0,0,0,0,0,1,1,1,1,0,1,1,1,0,1,1,1,0,1]
            case(2)                                                         ! POST
                allocate(opt_args(8), inc_args(8))
                opt_args = ''
                opt_args(7) = '--swap_angles'
                opt_args(8) = '--compare_tor_pos'
                inc_args(7:8) = [0,2]
        end select

        ! set common option arguments
        ltest = .false.                                                     ! don't call the testing routines
        lvl = 1
        opt_args(1) = '-t'
        opt_args(2) = '--test'
        opt_args(3) = '--no_plots'
        opt_args(4) = '--no_output'
        opt_args(5) = '--do_execute_command_line'
        opt_args(6) = '--mem_info'
        inc_args(1:6) = [0,0,0,0,0,0]
    end subroutine init_files

    !> Parses the command line arguments.
    !!
    !! \note The input arguments are saved in \c command_arg
    !!
    !! \return ierr
    integer function parse_args() result(ierr)
        use num_vars, only: prog_style, prog_name

        character(*), parameter :: rout_name = 'parse_args'

        ! local variables
        character(len=max_str_ln), allocatable :: open_error(:)             ! message for incorrect usage
        character(len=max_str_ln), allocatable :: open_help(:)              ! message for help
        integer :: iseq                                                     ! control the user-specified arguments
        integer :: id                                                       ! dummy integer
        character(len=max_str_ln) :: dum_str                                ! dummy string
        integer :: min_args                                                 ! minimal nr. of arguments

        ! initialize ierr
        ierr = 0

        ! Messages for the user
        ! select according to program style
        select case (prog_style)
            case(1)                                                         ! PB3D
                allocate(open_error(3))
                allocate(open_help(6))
                open_error(1) = ""                                          ! incorrect usage
                open_error(2) = "Usage: "//prog_name//&
                    &" USER_INPUT EQUILIBRIUM_INPUT [OPTIONS]"
                open_error(3) = "Try './"//prog_name//" --help' or './"//&
                    &prog_name//" -h' for more information."
                open_help(1) = open_error(2)                                 ! help with usage
                open_help(2) = ""
                open_help(3) = "    USER_INPUT          input provided by the &
                    &user"
                open_help(4) = "    EQUILIBRIUM_INPUT   output from VMEC or &
                    &HELENA, in .txt format"
                open_help(5) = "    [OPTIONS]           (optional) &
                    &command-line options"
                open_help(6) = ""
```

```
                min_args = 2
            case(2)                                                         ! POST
                allocate(open_error(3))
                allocate(open_help(6))
                open_error(1) = ""                                          ! incorrect usage
                open_error(2) = "Usage: "//prog_name//&
                    &" USER_INPUT PB3D_OUTPUT [OPTIONS]"
                open_error(3) = "Try './"//prog_name//" --help' or './"//&
                    &prog_name//" -h' for more information."
                open_help(1) = open_error(2)                                 ! help with usage
                open_help(2) = ""
                open_help(3) = "    USER_INPUT          input provided by the &
                    &user"
                open_help(4) = "    PB3D_OUTPUT    output from PB3D code, &
                    &in .h5 format"
                open_help(5) = "    [OPTIONS]           (optional) &
                    &command-line options"
                open_help(6) = ""
                min_args = 2
        end select

        call writo("Parsing command line arguments")

        ! Find number of arguments and first argument
        numargs = command_argument_count()
        dum_str = ''
        if (numargs.gt.0) call getarg(1,dum_str)

        ! print error if no arguments
        if (dum_str.eq.'-h' .or. dum_str.eq.'--help') then
            do id = 1,size(open_help)
                call writo(open_help(id))
            end do
            ierr = 66                                                       ! silent stop
            chckerr('')
        else if (numargs.lt.min_args) then
            do id = 1,size(open_error)
                call writo(open_error(id))
            end do
            ierr = 66                                                       ! silent stop
            chckerr('')
        else
            allocate(command_arg(numargs))
            command_arg = ""
            command_arg(1) = dum_str
        end if

        ! Get the rest of the arguments
        do iseq = 2, numargs
            call getarg(iseq, command_arg(iseq))
        end do

        call writo("Command line arguments parsed")
    end function parse_args

    !> Open the input files.
    !!
    !!  - input file with user options
    !!  - equilibrium file in
    !!      - NetCDF for VMEC
    !!      - plain for HELENA
    !!
    !! \return ierr
    integer function open_input() result(ierr)
        use num_vars, only: eq_i, input_i, rank, prog_style, no_plots, &
            &eq_style, eq_name, no_output, pb3d_i, pb3d_name, input_name, &
            &do_execute_command_line, output_name, prog_name, print_mem_usage, &
            &swap_angles, jump_to_sol, export_hel, plot_vmec_modes, &
            &compare_tor_pos, invert_top_bottom_h
        use rich_vars, only: no_guess
#if ldebug
        use num_vars, only: ltest
#endif

        character(*), parameter :: rout_name = 'open_input'

        ! local variables
        integer :: id                                                       ! counter
        character(len=max_str_ln) :: err_msg                                ! error message
        integer :: first_int                                                ! first input integer of input file (HELENA)
        integer :: istat                                                    ! status
        character(len=max_str_ln) :: file_ext                               ! file extension

        ! initialize ierr
        ierr = 0

        if (rank.eq.0) then                                                 ! following only for master process
```

```
            call writo("Opening files")
            call lvl_ud(1)

            ! select depending on program style
            select case (prog_style)
                case(1)                                                 ! PB3D
                    ! check for correct input file and use default if needed
                    input_name = command_arg(1)                         ! first argument is the name of the input file
                    open(unit=input_i,file=input_name,status='old',iostat=istat)
                    if (istat.eq.0) then
                        call writo('Input file "' // trim(input_name) &
                            &// '" opened')
                    else
                        call writo('No input file found. Default used')
                        input_name = ''
                    end if

                    ! check for  equilibrium file and  print error if  not found
                    ! (no default!)
                    eq_name = command_arg(2)
                    open(unit=eq_i,file=eq_name,status='old',iostat=ierr)
                    err_msg = 'No equilibrium file found.'
                    chckerr(err_msg)

                    ! succeeded
                    call writo('equilibrium file "' // trim(eq_name) &
                        &// '" opened')

                    ! Determine which equilibrium style (1: VMEC, 2: HELENA)
                    ! set eq_style to a nonsensical value
                    eq_style = -1

                    ! Check for VMEC
                    file_ext = ''
                    do id = len(eq_name)-1,1,-1
                        if (eq_name(id:id).eq.'.') then
                            file_ext = eq_name(id+1:len(eq_name))
                            exit
                        end if
                    end do
                    if (trim(file_ext).eq.'nc') then
                        eq_style = 1
                    end if

                    ! Check for HELENA
                    read(eq_i,*,iostat=istat) first_int
                    backspace(unit=eq_i)                                ! go back one line in the equilibrium file
                    if (istat.eq.0) then
                        eq_style = 2
                    end if

                    ! Check if something was found
                    if (eq_style.lt.1 .or. eq_style.gt.2) then
                        ierr = 1
                        call writo('Unable to recognize type of input file')
                        call writo('If you are trying to open HELENA files, &
                            &you need to have a version that outputs the &
                            &variable IAS. See tutorial!')
                        chckerr('')
                    end if

                    ! set PB3D output name
                    pb3d_name = prog_name//'_'//trim(output_name)//'.h5'
                case(2)                                                 ! POST
                    ! check for correct input file and use default if needed
                    input_name = command_arg(1)                         ! first argument is the name of the input file
                    open(unit=input_i,file=input_name,status='old',iostat=istat)
                    if (istat.eq.0) then
                        call writo('Input file "' // trim(input_name) &
                            &// '" opened')
                    else
                        call writo('No input file found. Default used')
                        input_name = ''
                    end if

                    ! check for PB3D_PB3D file and  print error if not found (no
                    ! default!)
                    pb3d_name = command_arg(2)
                    open(unit=pb3d_i,file=pb3d_name,status='old',iostat=ierr)
                    err_msg = 'No PB3D file found.'
                    chckerr(err_msg)

                    ! succeeded
                    call writo('PB3D output file "' // trim(pb3d_name) &
                        &// '" opened')
```

```
                    close(pb3d_i)
            end select

            ! set options
            if (numargs.gt.2) then                                          ! options given
                call writo('applying options')
                call lvl_ud(1)
                call read_opts
                call lvl_ud(-1)
            end if
            call lvl_ud(-1)
            call writo('Files opened')
        end if
    contains
        ! this subroutine scans for chosen options
        !> \private
        subroutine read_opts
#if ( lwith_intel && !lwith_gnu)
            use ifport
#endif
            integer :: jd
            integer :: numopts
            logical :: opt_taken(size(opt_args))                            ! which of the options has been taken
            logical :: opt_found

            ! set number of options still to be processed, no option taken yet
            numopts = size(opt_args)
            opt_taken = .false.
            opt_found = .false.

            id = 3                                                          ! start at third command argument
            do while (id.le.numargs)                                        ! try the remaining possible options
                jd = 1
                do while (jd.le.numopts .and. .not.opt_found)               ! try all the options
                    if (trim(command_arg(id)).eq.trim(opt_args(jd))) then   ! option found
                        opt_found = .true.
                        if (opt_taken(jd)) then                             ! option already taken
                            call writo('Option "'//trim(command_arg(id))//&
                                &'" already set, ignoring...',warning=.true.)
                        else                                                ! option not yet taken
                            select case(jd)
                                ! common options 1..6
                                case (1,2)                                  ! option test
#if ldebug
                                    call writo('option test chosen')
                                    ltest = .true.
#else
                                    call writo('option test not available. &
                                        &Recompile with cpp flag &
                                        &''ldebug''...',warning=.true.)
#endif
                                case (3)                                    ! disable plotting
                                    call writo('option no_plots chosen: &
                                        &plotting disabled')
                                    no_plots = .true.
                                case (4)                                    ! disable messages
                                    call writo('option no_output chosen: &
                                        &messages disabled')
                                    no_output = .true.
                                case (5)                                    ! disable execute_command_line
                                    call writo('option do_execute_command_line &
                                        &chosen: execute_command_line enabled')
                                    do_execute_command_line = .true.
                                case (6)                                    ! disable execute_command_line
#if ldebug
                                    call writo('option mem_info chosen: &
                                        &memory usage is printed')
                                    call lvl_ud(1)
                                    call writo('The PID of this process is: '//&
                                        &trim(i2str(getpid())))
                                    print_mem_usage = .true.
                                    call lvl_ud(-1)
#else
                                    call writo('option mem_info is only &
                                        &available when compiled with &
                                        &"ldebug"',warning=.true.)
#endif
                                ! specific options for each program style
                                case default
                                    select case (prog_style)
                                        case(1)                             ! PB3D
                                            call apply_opt_pb3d(jd,id)
                                        case(2)                             ! POST
                                            call apply_opt_post(jd,id)
                                    end select
                            end select
                            opt_taken(jd) = .true.
```

```
                        end if
                        id = id + inc_args(jd)                                  ! skip a number of next arguments
                    end if
                    jd = jd + 1
                    ! FIND THE INCREMENT WITH BELOW FUNCTION
                end do
                if (.not.opt_found) then
                    call writo('Option "'//trim(command_arg(id))//'" invalid',&
                        &warning=.true.)
                end if
                id = id + 1
                opt_found = .false.
            end do
        end subroutine read_opts

        ! apply chosen options for PB3D
        !> \private
        subroutine apply_opt_pb3d(opt_nr,arg_nr)                                ! PB3D version
            use num_vars, only: n_procs

            ! input / output
            integer, intent(in) :: opt_nr                                       ! option number
            integer, intent(in) :: arg_nr                                       ! argument number

            ! local variables
            integer :: id                                                       ! counter
            character(len=max_str_ln) :: opt_str                                ! string with option information

            select case(opt_nr)
                case (7)                                                        ! disable guessing Eigenfunction from previous Richardson level
                    call writo('option no_guess chosen: Eigenfunction not &
                        &guessed from previous Richardson level')
                    no_guess = .true.
                case (8)                                                        ! skip calculating perturbation variables
                    call writo('option jump_to_sol chosen: Skip all possible &
                        &equilibrium and perturbation drivers for first &
                        &Richardson level')
                    jump_to_sol = .true.
                case (9)                                                        ! export HELENA
                    if (eq_style.eq.2) then
                        if (n_procs.eq.1) then
                            call writo('option export_HEL chosen: Exporting to &
                                &VMEC')
                            export_hel = .true.
                        else
                            call writo('Can only use export_HEL with 1 &
                                &process',warning=.true.)
                            export_hel = .false.
                        end if
                    else
                        call writo('Can only use export_HEL with HELENA',&
                            &warning=.true.)
                        export_hel = .false.
                    end if
                case (10)                                                       ! output VMEC modes
                    if (eq_style.eq.1) then
                        call writo('option plot_VMEC_modes chosen: Plotting &
                            &(decay of) modes')
                        plot_vmec_modes = .true.
                    else
                        call writo('Can only output VMEC modes with VMEC',&
                            &warning=.true.)
                        plot_vmec_modes = .false.
                    end if
                case (11)                                                       ! invert top and bottom for HELENA
#if ldebug
                    if (eq_style.eq.2) then
                        call writo('option invert_top_bottom_H chosen: &
                            &inverting top and bottom of equilibrium')
                        invert_top_bottom_h = .true.
                    else
                        call writo('Can only invert top and bottom for HELENA',&
                            &warning=.true.)
                        invert_top_bottom_h = .false.
                    end if
#else
                    call writo('Can only invert top and bottom for HELENA in &
                        &debug mode',warning=.true.)
                    invert_top_bottom_h = .false.
#endif
                case (12:25)
                    opt_str = ''
                    do id = 1,inc_args(opt_nr)
                        opt_str = trim(opt_str)//' '//&
                            &trim(command_arg(arg_nr+id))
                    end do
```

```
                    call writo('option "'//trim(opt_args(opt_nr))//&
                        &trim(opt_str)//'" passed to SLEPC')
                case default
                    call writo('Invalid option number',warning=.true.)
            end select
        end subroutine apply_opt_pb3d

        ! apply chosen options for POST
        !> \private
        subroutine apply_opt_post(opt_nr,arg_nr)                                ! POST version
            use num_vars, only: rz_0

            ! input / output
            integer, intent(in) :: opt_nr                                       ! option number
            integer, intent(in) :: arg_nr                                       ! argument number

            ! local variables
            integer :: id                                                       ! counter

            select case(opt_nr)
                case (7)                                                        ! disable guessing Eigenfunction from previous Richardson level
                    call writo('option swap_angles chosen: theta and zeta are &
                        &swapped in plots')
                    swap_angles = .true.
                case (8)                                                        ! disable guessing Eigenfunction from previous Richardson level
                    call writo('option compare_tor_pos chosen: Comparing &
                        &B, J and kappa at different toroidal positions')
                    call lvl_ud(1)
                    compare_tor_pos = .true.
                    do id = 1,inc_args(opt_nr)
                        if (arg_nr+id.gt.size(command_arg)) then
                            ierr = 1
                        else
                            read(command_arg(arg_nr+id),*,iostat=ierr) rz_0(id)
                        end if
                        if (ierr.ne.0) then
                            call writo('Did you provide an origin for the &
                                &geometrical poloidal angle in the form')
                            call writo('"--compare_tor_pos R_0, Z_0"?')
                        end if
                        chckerr('failed to read RZ_0')
                    end do
                    call writo('Origin for geometrical poloidal angle: ('//&
                        &trim(r2str(rz_0(1)))//','//trim(r2str(rz_0(2)))//')')
                    call lvl_ud(-1)
                case default
                    call writo('Invalid option number',warning=.true.)
            end select
        end subroutine apply_opt_post
    end function open_input

    !> Open the output files
    !!
    !!  - output file <tt>.txt</tt>
    !!  - shell commands file <tt>.sh</tt>
    !!  - HDF5 file: only for PB3D, not for POST.
    !!  - memory usage file <tt>.dat</tt>
    !!
    !! Also sets some output variables.
    !!
    !! \note
    !!  -# memory usage file is only for debug version.
    !!  -# There can be resart of a Richardson level for PB3D
    !!  -# There  can also be  a direct  jump to the  solution for PB3D,  if the
    !!  equilibrium  and perturbation  phases are  already done  and saved  (see
    !!  init_files()).
    !!  -# In the case of a Richardson restart, PB3D reopens the HDF5 file.
    !!
    !! \return ierr
    integer function open_output() result(ierr)
        use num_vars, only: prog_style, output_i, output_name, prog_name, &
            &rich_restart_lvl, shell_commands_name, shell_commands_i, &
            &pb3d_name, jump_to_sol
        use hdf5_ops, only: create_output_hdf5
        use hdf5_utilities, only: probe_hdf5_group
#if ldebug
        use num_vars, only: print_mem_usage, mem_usage_name, mem_usage_i
#endif
#if ( lwith_intel && !lwith_gnu)
        use ifport
#endif

        character(*), parameter :: rout_name = 'open_output'

        ! local variables (also used in child functions)
```

```
        integer :: id                                                   ! counter
        integer :: istat                                                ! status
        logical :: file_exists                                          ! whether file exists
        logical :: group_exists                                         ! whether probed group exists
        character(len=max_str_ln) :: full_output_name                   ! full name
        character(len=max_str_ln) :: err_msg                            ! error message

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Attempting to open output files')
        call lvl_ud(1)

        ! 1. LOG OUTPUT

        ! append extension to output name
        full_output_name = prog_name//'_'//trim(output_name)//'.txt'

        ! actions depending on Richardson restart level and program style
        if (rich_restart_lvl.gt.1 .and. prog_style.eq.1) then           ! PB3D restart
            ! append to existing file
            open(output_i,file=trim(full_output_name),status='old',&
                &position='append',iostat=ierr)
            chckerr('Failed to open output file')

            ! print message
            call writo('log output file "'//trim(full_output_name)//&
                &'" reopened at number '//trim(i2str(output_i)))
        else                                                            ! POST or PB3D no restart
            ! open file after wiping it
            open(output_i,file=trim(full_output_name),status='replace',&
                &iostat=ierr)
            chckerr('Failed to open output file')

            ! print message
            call writo('log output file "'//trim(full_output_name)//&
                &'" opened at number '//trim(i2str(output_i)))
        end if

        ! if temporary output present, silently write it to log output file
        do id = 1,size(temp_output)
            write(output_i,'(A)',iostat=istat) trim(temp_output(id))
        end do

        ! 2. SHELL COMMANDS

        ! recycle full_output_name for shell_commands file
        full_output_name = prog_name//'_'//trim(shell_commands_name)//'.sh'

        ! create output file for shell commands
        open(unit=shell_commands_i,file=trim(full_output_name),&
            &status='replace',iostat=ierr)
        chckerr('Failed to create shell command file')

        ! write header, close and make executable
        write(shell_commands_i,'(A)',iostat=istat) '#!/bin/bash'
        write(shell_commands_i,'(A)',iostat=istat) '# This file contains all &
            &the shell commands from the '//trim(prog_name)//' run'
        close(shell_commands_i)
        istat = 0
#if ( lwith_intel && !lwith_gnu)
        istat = system('chmod +x '//trim(full_output_name))
#else
        call execute_command_line('chmod +x '//trim(full_output_name),&
            &exitstat=istat)                                            ! not too terrible if execute_command_line fails
#endif

        ! print message
        call writo('shell commands script file "'//trim(full_output_name)//&
            &'" created')

        ! 3. PROGRAM SPECIFIC

        ! specific actions for program styles
        select case (prog_style)
            case (1)                                                    ! PB3D
                ! set up whether group exists
                inquire(file=pb3d_name,exist=file_exists,iostat=ierr)
                chckerr('Failed to inquire about file')
                if (file_exists) then
                    ierr = probe_hdf5_group(pb3d_name,'X_2_int_R_'//&
                        &trim(i2str(rich_restart_lvl)),group_exists)
                    chckerr('')
                else
                    group_exists = .false.
```

```
                end if

                if (jump_to_sol) then
                    if (.not.group_exists) then
                        ierr = 1
                        call writo('No integrated tensorial perturbation &
                            &quantitites found to jump over',alert=.true.)
                        err_msg = 'These quantitites need to be set up'
                        chckerr(err_msg)
                    end if
                else
                    ! give warning
                    if (group_exists) then
                        call writo('Integrated tensorial perturbation &
                            &quantitites were found for Richardson level '&
                            &//trim(i2str(rich_restart_lvl))//'.',&
                            &alert=.true.)
                        call lvl_ud(1)
                        call writo('Have you already calculated them and &
                            &are you re-running these simulations')
                        call writo('because you want to change the solution?')
                        call writo('If the solution method is the only &
                            &thing you are changing, you should')
                        call writo('consider using the command-line option &
                            &"--jump_to_sol"')
                        call lvl_ud(-1)
                    end if

                    ! create HDF5 file for output if no restart
                    if (rich_restart_lvl.eq.1) then
                        ierr = create_output_hdf5(pb3d_name)
                        chckerr('')
                    end if
                end if
            case (2)                                                    ! POST
                ! do nothing
        end select

#if ldebug
        ! 4. MEMORY USAGE
        if (print_mem_usage) then
            ! recycle full_output_name for mem_usage file
            full_output_name = prog_name//'_'//trim(mem_usage_name)//'.dat'

            ! create output file for memory usage
            open(mem_usage_i,file=trim(full_output_name),status='replace',&
                &iostat=ierr)
            chckerr('Failed to open memory file')

            ! write header and close
            write(mem_usage_i,'(A)',iostat=istat) '#   Rank []  Count [] &
                &            Time [s] Mem. [kB]  Max. tot. Memory [kB] &
                &Max. X. Memory [kB]'
            close(mem_usage_i)

            ! print message
            call writo('memory usage data file "'//trim(full_output_name)//&
                &'" created')
        end if
#endif

        ! no more temporary output
        temp_output_active = .false.

        ! deallocate temporary output if allocated
        if (allocated(temp_output)) deallocate(temp_output)

        call lvl_ud(-1)
        call writo('Output files opened')
    end function open_output

    !> Closes the output file.
    subroutine close_output()
        use num_vars, only: rank, output_i, prog_name, shell_commands_name, &
            &do_execute_command_line

        call writo('Closing output files')
        call lvl_ud(1)
        call writo('A log of the shell command is kept in the log file "'//&
            &trim(prog_name)//'_'//trim(shell_commands_name)//'.sh"')
        if (.not.do_execute_command_line) &
            &call writo('Execute this script to do all the shell commands')
        call lvl_ud(-1)
        call writo('Output files closed')
        call writo('')

        if (rank.eq.0) close(output_i)
```

```
    end subroutine
end module files_ops
```

D.26 /Users/toonweyens/Code/PB3D/Modules/files_utilities.f90 File Reference

Modules

- module files_utilities

  Numerical utilities related to files.

Functions/Subroutines

- integer function, public files_utilities::nextunit (unit)

  Search for available new unit.
- integer function, public files_utilities::skip_comment (file_i, file_name)

  Skips comment when reading a file.
- subroutine, public files_utilities::get_file_info (file_name, file_size, acc_time, mod_time)

  Gets file information.
- character(len=max_str_ln) function, public files_utilities::get_full_pb3d_name (rich_lvl)

  Returns the name of the PB3D output file.
- integer function, public files_utilities::delete_file (file_name)

  Removes a file.
- integer function, public files_utilities::count_lines (file_i)

  Count non-comment lines in a file.

D.27 files_utilities.f90

Go to the documentation of this file.

```
!------------------------------------------------------------------------!
!> Numerical utilities related to files
!------------------------------------------------------------------------!
module files_utilities
#include <PB3D_macros.h>
    use str_utilities
    use messages
    use num_vars, only: dp, max_str_ln
    implicit none
    private
    public nextunit, get_file_info, get_full_pb3d_name, delete_file, &
        &skip_comment, count_lines

contains
    !> Search for available new unit.
    !!
    !! \c lun_min and \c lun_max define the range of possible luns to check.
    !!
    !! The unit  value is  returned by  the function, and  also by  the optional
    !! argument.  This allows  the  function  to be  used  directly  in an  open
    !! statement, and optionally save the result in a local variable.
    !!
    !! If no units are available, -1 is returned.
    !!
    !! \see Adapted from http://fortranwiki.org/fortran/show/newunit
    integer function nextunit(unit)
        ! input / output
        integer, intent(out), optional :: unit
```

```fortran

        ! local variables
        integer, parameter :: lun_min=70, lun_max=1000
        logical :: file_open
        integer :: lun

        ! iterate over permitted luns until available one found
        nextunit=-1
        do lun=lun_min,lun_max
            inquire(unit=lun,opened=file_open)
            if (.not.file_open) then
                nextunit=lun
                exit
            end if
        end do

        ! return unit number if present
        if (present(unit)) unit=nextunit
    end function nextunit

    !> Skips comment when reading a file.
    !!
    !! By comment, a line is meant that starts with the character \c \#.
    !!
    !! \return ierr
    integer function skip_comment(file_i,file_name) result(ierr)
        character(*), parameter :: rout_name = 'skip_comment'

        ! input / output
        integer, intent(in) :: file_i                                   ! file identifier
        character(len=*), intent(in), optional :: file_name             ! file name

        ! local variables
        character(len=1) :: loc_data_char                               ! first data character
        character(len=max_str_ln) :: err_msg                            ! error message

        ! initialize ierr
        ierr = 0

        loc_data_char = '#'
        do while (loc_data_char.eq.'#')
            read(file_i,*,iostat=ierr) loc_data_char
            if (present(file_name)) then
                err_msg = 'failed to read "'//trim(file_name)//'"'
            else
                err_msg = 'failed to read file'
            end if
            chckerr(err_msg)
        end do
        backspace(file_i)
    end function skip_comment

    !> Gets file information.
    !!
    !! The time  informations can  be converted to  strings using  the intrinsic
    !! function "ctime".
    subroutine get_file_info(file_name,file_size,acc_time,mod_time)
        ! input / output
        character(len=*), intent(in) :: file_name                       !< name of file
        integer, intent(inout), optional :: file_size                   !< file size
        integer, intent(inout), optional :: acc_time                    !< file access time
        integer, intent(inout), optional :: mod_time                    !< file modification time

        ! local variables
        integer :: vals(13)                                             ! values
        integer :: istat                                                ! status

        ! call stat
        call stat(trim(file_name),vals,istat)

        ! check status
        if (istat.ne.0) then
            call writo('call to stat failed',warning=.true.)
            if (present(file_size)) file_size = 0
            if (present(acc_time)) acc_time = 0
            if (present(mod_time)) mod_time = 0
        else
            if (present(file_size)) file_size = vals(8)
            if (present(acc_time)) acc_time = vals(9)
            if (present(mod_time)) mod_time = vals(10)
        end if
    end subroutine get_file_info

    !> Returns the name of the PB3D output file.
    !!
    !! Optionally, the Richardson level can be appended as <tt>_R_[lvl]</tt>.
    !!
```

```
    !! If not positive, it is ignored.
    character(len=max_str_ln) function get_full_pb3d_name(rich_lvl) &
        &result(full_pb3d_name)
        use num_vars, only: pb3d_name

        ! input / output
        integer, intent(in), optional :: rich_lvl                           ! optional richardson level to be appended

        ! local variables
        integer :: rich_lvl_loc                                             ! local richardson level

        ! set local rich_lvl
        rich_lvl_loc = 0
        if (present(rich_lvl)) rich_lvl_loc = rich_lvl

        ! set ouput
        if (rich_lvl_loc.gt.0) then
            full_pb3d_name = trim(pb3d_name)//'_R_'//&
                &trim(i2str(rich_lvl_loc))//'.h5'
        else
            full_pb3d_name = trim(pb3d_name)//'.h5'
        end if
    end function get_full_pb3d_name

    !> Removes a file.
    !!
    !! \return istat
    integer function delete_file(file_name) result(istat)
        ! input / output
        character(len=*), intent(inout) :: file_name                        !< the name that is deleted

        ! local variables
        integer :: file_i                                                   ! file unit
        logical :: file_open                                                ! whether file is open

        ! initialize istat
        istat = 0

        ! check if opened and open if not
        inquire(file=trim(file_name),opened=file_open,number=file_i,&
            &iostat=istat)
        chckstt
        if (.not.file_open) then
            open(unit=nextunit(file_i),file=trim(file_name),iostat=istat)
        end if
        chckstt

        ! remove open file
        close(unit=file_i,status='delete',iostat=istat)
        chckstt
    end function delete_file

    !> Count non-comment lines in a file.
    integer function count_lines(file_i) result(nr_lines)
        ! input / output
        integer, intent(in) :: file_i                                       !< file identifier

        ! local variables
        integer :: istat                                                    ! status
        character(len=1) :: loc_data_char                                   ! first data character

        ! initialize istat
        istat = 0

        nr_lines = 0
        rewind(file_i)
        do while (istat.eq.0)
            ! read first character of data
            read(file_i,*,iostat=istat) loc_data_char
            if (istat.eq.0 .and. loc_data_char.ne.'#') then                 ! exclude comment lines
                nr_lines = nr_lines + 1
            end if
        end do
        rewind(file_i)
    end function count_lines
end module files_utilities
```

D.28 /Users/toonweyens/Code/PB3D/Modules/grid_ops.f90 File Reference

Modules

- module grid_ops

  Operations that have to do with the grids and different coordinate systems.

Functions/Subroutines

- integer function, public grid_ops::calc_norm_range (style, in_limits, eq_limits, x_limits, sol_limits, r_f↩_eq, r_f_x, r_f_sol, jq)

  Calculates normal range for the input grid, the equilibrium grid and/or the solution grid.
- integer function calc_norm_range_pb3d_in (in_limits)

  PB3D input version.
- subroutine calc_norm_range_pb3d_eq (eq_limits)

  PB3D equilibrium version.
- integer function calc_norm_range_pb3d_x (eq_limits, x_limits, r_f_eq, r_f_x, jq)

  PB3D perturbation version.
- integer function calc_norm_range_pb3d_sol (sol_limits, r_f_sol, n_procs)

  PB3D solution version.
- subroutine calc_norm_range_post (eq_limits, x_limits, sol_limits, r_f_eq, r_f_x, r_f_sol)

  POST version.
- integer function, public grid_ops::setup_grid_eq (grid_eq, eq_limits)

  Sets up the equilibrium grid.
- integer function, public grid_ops::setup_grid_eq_b (grid_eq, grid_eq_b, eq, only_half_grid)

  Sets up the field-aligned equilibrium grid.
- integer function, public grid_ops::setup_grid_x (grid_eq, grid_x, r_f_x, x_limits)

  Sets up the general perturbation grid, in which the perturbation variables are calculated.
- integer function, public grid_ops::setup_grid_sol (grid_eq, grid_x, grid_sol, r_f_sol, sol_limits)

  Sets up the general solution grid, in which the solution variables are calculated.
- integer function, public grid_ops::calc_ang_grid_eq_b (grid_eq, eq, only_half_grid)

  Calculate equilibrium grid that follows magnetic field lines.
- integer function, public grid_ops::redistribute_output_grid (grid, grid_out, no_outer_trim)

  Redistribute a grid to match the normal distribution of solution grid.
- integer function, public grid_ops::magn_grid_plot (grid)

  Plots the grid in real 3-D space.
- integer function, public grid_ops::print_output_grid (grid, grid_name, data_name, rich_lvl, par_div, remove_previous_arrs)

  Print grid variables to an output file.

Variables

- logical, public grid_ops::debug_calc_ang_grid_eq_b = .false.

  plot debug information for calc_ang_grid_eq_b()

D.28.1 Function/Subroutine Documentation

D.28.1.1 calc_norm_range_pb3d_eq()

```
subroutine calc_norm_range::calc_norm_range_pb3d_eq (
            integer, dimension(2), intent(inout) eq_limits)
```

PB3D equilibrium version.

The normal range is calculated by dividing the full equilibrium range passed from the input phase between the processes.

Note

For X_style 2 (solution) or 3 (optimized), at the end of the equilibrium phase, there will be an exchange of data from each process to each process so that they all have the tightest possible fit of data of the corresponding perturbation limits.

Parameters

in,out	*eq_limits*	min. and max. index of eq. grid for this process

Definition at line 245 of file grid_ops.f90.

Here is the caller graph for this function:

D.28.1.2 calc_norm_range_pb3d_in()

```
integer function calc_norm_range::calc_norm_range_pb3d_in (
            integer, dimension(2), intent(inout) in_limits)
```

PB3D input version.

The normal range is calculated by finding the tightest range of the input variables encompassing the entire solution range. Additionally, the global max_flux variables are set as well..

Note

This is the global, undivided range. The division information is calculated in 'eq_limits'. Furthermore, the input variables have to be tabulated on the full grid provided by the equilibrium code to be able to be used in PB3D and POST.

Parameters

in,out	*in_limits*	total min. and max. index of eq. grid for this process

Definition at line 135 of file grid_ops.f90.

Here is the caller graph for this function:

D.28.1.3 calc_norm_range_pb3d_sol()

```
integer function calc_norm_range::calc_norm_range_pb3d_sol (
                integer, dimension(2), intent(inout) sol_limits,
                real(dp), dimension(:), intent(inout) r_f_sol,
                integer, intent(in), optional n_procs)
```

PB3D solution version.

The normal range is determined by simply dividing the solution range, a ghost range is required.

By default, this routine uses `sol_n_procs` processes, but this can be overruled.

Parameters

in,out	*sol_limits*	min. and max. index of sol grid for this process
in,out	*r_f_sol*	solution r_F
in	*n_procs*	how many processes used

Definition at line 386 of file grid_ops.f90.

Here is the caller graph for this function:

D.28.1.4 calc_norm_range_pb3d_x()

```
integer function calc_norm_range::calc_norm_range_pb3d_x (
            integer, dimension(2), intent(in) eq_limits,
            integer, dimension(2), intent(inout) x_limits,
            real(dp), dimension(:), intent(in) r_f_eq,
            real(dp), dimension(:), intent(inout), allocatable r_f_x,
            real(dp), dimension(:), intent(in) jq)
```

PB3D perturbation version.

The normal range is determined according to X_grid style:

1. taken identical to the equilibrium grid, which means that after the perturbation phase the integrated tensorial quantities have to be interpolated in interp_v() in driver_sol() after having been redistributed at the start of the solution driver.
2. taken identical to the solution grid, which means that the the equilibrium quantities have to be interpolated in calc_u(), calc_kv() and calc_pv() after having been redistributed at the end of the equilibrium driver.
3. by considering an optimal interpolation extension of the equilibrium range, adding intermediairy points between grid points where the safety factor changes too quickly, according to the variable 'max_njq_change'. This uses `prim_X`.

Parameters

`in`	*eq_limits*	min. and max. index of eq grid for this process
`in,out`	*x_limits*	min. and max. index of X grid for this process
`in`	*r_f_eq*	equilibrium r_F
`in,out`	*r_f_x*	perturbation r_F
`in`	*jq*	q_saf (pol. flux) or rot_t (tor. flux) in total Flux variables

Definition at line 283 of file grid_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

D.28.1.5 calc_norm_range_post()

```
subroutine calc_norm_range::calc_norm_range_post (
            integer, dimension(2), intent(inout) eq_limits,
            integer, dimension(2), intent(inout) x_limits,
            integer, dimension(2), intent(inout) sol_limits,
            real(dp), dimension(:), intent(in) r_f_eq,
            real(dp), dimension(:), intent(in) r_f_x,
            real(dp), dimension(:), intent(in) r_f_sol)
```

POST version.

The normal range is determined by simply dividing a possible subset of the solution range, indicated by `min_r_plot` and `max_r_sol`, including a ghost range and getting a bounding equilibrium and perturbation range.

Parameters

`in,out`	*eq_limits*	min. and max. index of eq grid for this process

in,out	*x_limits*	min. and max. index of X grid for this process
in,out	*sol_limits*	min. and max. index of sol grid for this process
in	*r_f_eq*	eq r_F
in	*r_f_x*	X r_F
in	*r_f_sol*	sol r_F

Definition at line 451 of file grid_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

D.29 grid_ops.f90

Go to the documentation of this file.

```
00001 !-----------------------------------------------------------------------!
00002 !> Operations that have to do with the grids and different coordinate systems.
00003 !-----------------------------------------------------------------------!
00004 module grid_ops
00005 #include <PB3D_macros.h>
00006     use str_utilities
00007     use output_ops
00008     use messages
00009     use num_vars, only: dp, pi, max_str_ln
00010     use grid_vars, only: grid_type
00011     use eq_vars, only: eq_1_type
00012
00013     implicit none
00014     private
00015     public calc_norm_range, calc_ang_grid_eq_b, magn_grid_plot, setup_grid_eq, &
```

```
        &setup_grid_eq_b, setup_grid_x, setup_grid_sol, print_output_grid, &
        &redistribute_output_grid
#if ldebug
    public debug_calc_ang_grid_eq_b
#endif

    ! global variables
#if ldebug
    !> \ldebug
    logical :: debug_calc_ang_grid_eq_b = .false.                          !< plot debug information for calc_ang_grid_eq_b()
#endif

contains
    !> Calculates normal range  for the input grid, the  equilibrium grid and/or
    !! the solution grid.
    !!
    !! General workings, depending on \c X_grid_style:
    !!
    !! | 1 (equilibrium)     | 2 (solution )       | 3 (enriched)           |
    !! | ------------------- | ------------------- | ---------------------- |
    !! | calc_eq()           | calc_eq()           | calc_eq()              |
    !! |                     | redistribute to sol | add points to optimize |
    !! |                     | interpolate to sol  | interpolate to X       |
    !! | calc_x()            | calc_x()            | calc_x()               |
    !! | redistribute to sol | copy to sol         | redistribute to sol    |
    !! | interpolate to sol  |                     | interpolate to sol     |
    !!
    !! \return ierr
    integer function calc_norm_range(style,in_limits,eq_limits,X_limits,&
        &sol_limits,r_F_eq,r_F_X,r_F_sol,jq) result(ierr)

        character(*), parameter :: rout_name = 'calc_norm_range'

        ! input / output
        character(len=*), intent(in) :: style                              !< style of calculation (PB3D: in, eq, X or sol; POST)
        integer, intent(inout), optional :: in_limits(2)                   !< min. and max. index of in grid
        integer, intent(inout), optional :: eq_limits(2)                   !< min. and max. index of eq grid for this process
        integer, intent(inout), optional :: x_limits(2)                    !< min. and max. index of X grid for this process
        integer, intent(inout), optional :: sol_limits(2)                  !< min. and max. index of sol grid for this process
        real(dp), intent(inout), optional :: r_f_eq(:)                     !< equilibrium r_F
        real(dp), intent(inout), allocatable, optional :: r_f_x(:)         !< perturbation r_F
        real(dp), intent(inout), optional :: r_f_sol(:)                    !< solution r_F
        real(dp), intent(in), optional :: jq(:)                            !< q_saf (pol. flux) or rot_t (tor. flux) in total Flux variables

        ! local variables
        character(len=max_str_ln) :: err_msg                               ! error message

        ! initialize ierr
        ierr = 0

        ! select depending on style
        select case (style)
            case ('PB3D_in')                                               ! PB3D: in
                if (present(in_limits)) then
                    ierr = calc_norm_range_pb3d_in(in_limits)
                    chckerr('')
                else
                    ierr = 1
                    err_msg = 'for PB3D: in, in_limits need to be provided'
                    chckerr(err_msg)
                end if
            case ('PB3D_eq')                                               ! PB3D: eq
                if (present(eq_limits)) then
                    call calc_norm_range_pb3d_eq(eq_limits)
                else
                    ierr = 1
                    err_msg = 'for PB3D: eq, eq_limits need to be provided'
                    chckerr(err_msg)
                end if
            case ('PB3D_X')                                                ! PB3D: X
                if (present(eq_limits).and.present(x_limits).and.&
                    &present(r_f_eq).and.present(r_f_x).and.(present(jq))) then
                    ierr = calc_norm_range_pb3d_x(eq_limits,x_limits,&
                        &r_f_eq,r_f_x,jq)
                    chckerr('')
                else
                    ierr = 1
                    err_msg = 'for PB3D: X, eq_limits, X_limits, r_F_eq, &
                        &r_F_X and jq need to be provided'
                    chckerr(err_msg)
                end if
```

```
            case ('PB3D_sol')                                           ! PB3D: sol
                if (present(sol_limits).and.present(r_f_sol)) then
                    ierr = calc_norm_range_pb3d_sol(sol_limits,r_f_sol)
                    chckerr('')
                else
                    ierr = 1
                    err_msg = 'for PB3D: sol, sol_limits and r_F_sol need to &
                        &be provided'
                    chckerr(err_msg)
                end if
            case ('POST')                                               ! POST
                if (present(eq_limits) .and. present(x_limits) .and. &
                    &present(sol_limits) .and.  present(r_f_eq) .and. &
                    &present(r_f_x) .and.  present(r_f_sol)) then
                    call calc_norm_range_post(eq_limits,x_limits,sol_limits,&
                        &r_f_eq,r_f_x,r_f_sol)
                else
                    ierr = 1
                    err_msg = 'for POST, eq_limits, X_limits, sol_limits, &
                        &r_F_eq, r_F_X and r_F_sol need to be provided'
                    chckerr(err_msg)
                end if
            case default
                ierr = 1
                err_msg = 'Incorrect style "'//trim(style)//'"'
                chckerr(err_msg)
        end select
    contains
        !> \public PB3D input version
        !!
        !! The normal range  is calculated by finding the tightest  range of the
        !! input variables encompassing the entire solution range. Additionally,
        !! the global max_flux variables are set as well..
        !!
        !! \note This is  the global, undivided range.  The division information
        !! is calculated  in 'eq_limits'. Furthermore, the  input variables have
        !! to be tabulated on the full  grid provided by the equilibrium code to
        !! be able to be used in PB3D and POST.
        integer function calc_norm_range_pb3d_in(in_limits) &
            &result(ierr)                                               ! PB3D version for equilibrium grid
            use num_vars, only: use_pol_flux_e, use_pol_flux_f, eq_style, &
                &norm_disc_prec_eq
            use num_utilities, only: con2dis, dis2con, round_with_tol, spline
            use vmec_vars, only: flux_p_v, flux_t_v
            use helena_vars, only: flux_p_h, flux_t_h
            use x_vars, only: min_r_sol, max_r_sol
            use eq_vars, only: max_flux_e, max_flux_f
            use grid_vars, only: n_r_in

            character(*), parameter :: rout_name = 'calc_norm_range_PB3D_in'

            ! input / output
            integer, intent(inout) :: in_limits(2)                      !< total min. and max. index of eq. grid for this process

            ! local variables
            real(dp), allocatable :: flux_f(:), flux_e(:)               ! either pol. or tor. flux in F and E
            real(dp) :: tot_min_r_in_f_con                              ! tot_min_r_in in continuous F coords.
            real(dp) :: tot_max_r_in_f_con                              ! tot_max_r_in in continuous F coords.
            real(dp) :: tot_min_r_in_e_con(1)                           ! tot_min_r_in in continuous E coords.
            real(dp) :: tot_max_r_in_e_con(1)                           ! tot_max_r_in in continuous E coords.
            real(dp) :: tot_min_r_in_e_dis                              ! tot_min_r_in in discrete E coords., unrounded
            real(dp) :: tot_max_r_in_e_dis                              ! tot_max_r_in in discrete E coords., unrounded

            ! initialize ierr
            ierr = 0

            ! set up flux_F and flux_E,  depending on which equilibrium style is
            ! being used:
            !   1:  VMEC
            !   2:  HELENA
            allocate(flux_f(n_r_in),flux_e(n_r_in))
            select case (eq_style)
                case (1)                                                ! VMEC
                    ! set up F flux
                    if (use_pol_flux_f) then
                        flux_f = flux_p_v(:,0)
                    else
                        flux_f = - flux_t_v(:,0)                        ! conversion VMEC LH -> RH coord. system
                    end if
                    ! set up E flux
                    if (use_pol_flux_e) then
                        flux_f = flux_p_v(:,0)
                    else
                        flux_e = flux_t_v(:,0)
```

```
                    end if
                case (2)                                                    ! HELENA
                    ! set up F flux
                    if (use_pol_flux_f) then
                        flux_f = flux_p_h(:,0)
                    else
                        flux_f = flux_t_h(:,0)
                    end if
                    ! set up E flux
                    if (use_pol_flux_e) then
                        flux_e = flux_p_h(:,0)
                    else
                        flux_e = flux_t_h(:,0)
                    end if
            end select

            ! set max_flux
            max_flux_e = flux_e(n_r_in)
            max_flux_f = flux_f(n_r_in)

            ! normalize flux_F and flux_E to (0..1) using max_flux
            flux_e = flux_e/max_flux_e
            flux_f = flux_f/max_flux_f

            ! get lower and upper bound of total solution range
            ! 1. start
            tot_min_r_in_f_con = min_r_sol
            tot_max_r_in_f_con = max_r_sol
            ! 2. round with tolerance
            ierr = round_with_tol(tot_min_r_in_f_con,0._dp,1._dp)
            chckerr('')
            ierr = round_with_tol(tot_max_r_in_f_con,0._dp,1._dp)
            chckerr('')
            ! 3. continuous E coords
            ierr = spline(flux_f,flux_e,[tot_min_r_in_f_con],&
                &tot_min_r_in_e_con,ord=norm_disc_prec_eq)
            ierr = spline(flux_f,flux_e,[tot_max_r_in_f_con],&
                &tot_max_r_in_e_con,ord=norm_disc_prec_eq)
            ! 4. round with tolerance
            ierr = round_with_tol(tot_min_r_in_e_con,minval(flux_e),&
                &maxval(flux_e))
            chckerr('')
            ierr = round_with_tol(tot_max_r_in_e_con,minval(flux_e),&
                &maxval(flux_e))
            chckerr('')
            ! 5. discrete E index, unrounded
            ierr = con2dis(tot_min_r_in_e_con(1),tot_min_r_in_e_dis,flux_e)
            chckerr('')
            ierr = con2dis(tot_max_r_in_e_con(1),tot_max_r_in_e_dis,flux_e)
            chckerr('')
            ! 6. discrete E index, rounded
            in_limits(1) = floor(tot_min_r_in_e_dis)
            in_limits(2) = ceiling(tot_max_r_in_e_dis)
        end function calc_norm_range_pb3d_in

        !> \public PB3D equilibrium version
        !!
        !! The normal range is calculated by dividing the full equilibrium range
        !! passed from the input phase between the processes.
        !!
        !! \note For  X_style 2 (solution) or  3 (optimized), at the  end of the
        !! equilibrium  phase, there  will  be  an exchange  of  data from  each
        !! process to each  process so that they all have  the tightest possible
        !! fit of data of the corresponding perturbation limits.
        subroutine calc_norm_range_pb3d_eq(eq_limits)                       ! PB3D version for equilibrium grid
            use num_vars, only: n_procs, rank, norm_disc_prec_eq
            use grid_vars, only: n_r_eq

            ! input / output
            integer, intent(inout) :: eq_limits(2)                          !< min. and max. index of eq. grid for this process

            ! set local equilibrium limits
            eq_limits(1) = nint(1 + 1._dp*rank*(n_r_eq-1)/n_procs)
            eq_limits(2) = nint(1 + (1._dp+rank)*(n_r_eq-1)/n_procs)

            ! increase lower limits for processes that are not first
            if (rank.gt.0) eq_limits(1) = eq_limits(1)+1

            ! ghost regions of width 4*norm_disc_prec_eq
            ! Note: When finite differences were used, a factor of 2 was enough,
            ! but it  looks like for  splines it has to  be higher. Even  for 4,
            ! there is some slight discrepancy...  This should be looked into in
            ! the future.
            eq_limits(1) = max(eq_limits(1)-4*norm_disc_prec_eq,1)
            eq_limits(2) = min(eq_limits(2)+4*norm_disc_prec_eq,n_r_eq)
        end subroutine calc_norm_range_pb3d_eq
```

```fortran

        !> \public PB3D perturbation version
        !!
        !! The normal range is determined according to X_grid style:
        !!  1. taken identical  to the equilibrium grid, which  means that after
        !!  the perturbation  phase the integrated tensorial  quantities have to
        !!  be  interpolated in  interp_v()  in driver_sol()  after having  been
        !!  redistributed at the start of the solution driver.
        !!  2.  taken identical  to  the  solution grid,  which  means that  the
        !!  the  equilibrium quantities  have  to be  interpolated in  calc_u(),
        !!  calc_kv() and calc_pv()  after having been redistributed  at the end
        !!  of the equilibrium driver.
        !!  3. by   considering  an   optimal  interpolation  extension  of  the
        !!  equilibrium range,  adding intermediairy points between  grid points
        !!  where  the  safety factor  changes  too  quickly, according  to  the
        !!  variable 'max_njq_change'. This uses \c prim_X.
        integer function calc_norm_range_pb3d_x(eq_limits,X_limits,r_F_eq,&
            &r_F_X,jq) result(ierr)                                         ! PB3D version for perturbation grid

            use num_vars, only: x_grid_style, max_njq_change, rank
            use grid_vars, only: n_r_eq, n_r_sol
            use x_vars, only: prim_x
            use grid_utilities, only: calc_eqd_grid
            use eq_vars, only: max_flux_f

            character(*), parameter :: rout_name = 'calc_norm_range_PB3D_X'

            ! input / output
            integer, intent(in) :: eq_limits(2)                             !< min. and max. index of eq grid for this process
            integer, intent(inout) :: x_limits(2)                           !< min. and max. index of X grid for this process
            real(dp), intent(in) :: r_f_eq(:)                               !< equilibrium r_F
            real(dp), intent(inout), allocatable :: r_f_x(:)                !< perturbation r_F
            real(dp), intent(in) :: jq(:)                                   !< q_saf (pol. flux) or rot_t (tor. flux) in total Flux variables

            ! local variables
            integer :: kd                                                   ! counter
            integer, allocatable :: div(:)                                  ! number of extra divisions for this grid interval
            real(dp), allocatable :: r_f_plot(:,:)                          ! plot of r_F_eq and r_F_X

            ! initialize ierr
            ierr = 0

            select case (x_grid_style)
                case (1)                                                    ! equilibrium
                    ! copy equilibrium
                    allocate(r_f_x(n_r_eq))
                    r_f_x = r_f_eq
                    x_limits = eq_limits

                    ! output
                    call writo('Using the equilibrium grid')
                case (2)                                                    ! solution
                    ! calculate solution
                    allocate(r_f_x(n_r_sol))
                    ierr = calc_norm_range_pb3d_sol(sol_limits=x_limits,&
                        &r_f_sol=r_f_x)
                    chckerr('')

                    ! output
                    call writo('Using the solution grid')
                case (3)                                                    ! enriched
                    ! decide extra divisions for each interval
                    allocate(div(size(r_f_eq)-1))
                    do kd = 1,size(r_f_eq)-1
                        div(kd) = &
                            &floor(prim_x*abs(jq(kd+1)-jq(kd))/max_njq_change)
                    end do

                    ! set up r_F_X with divisions
                    allocate(r_f_x(size(r_f_eq)+sum(div)))
                    do kd = 1,size(r_f_eq)-1
                        ierr = calc_eqd_grid(&
                            &r_f_x(kd+sum(div(1:kd-1)):kd+sum(div(1:kd))),&
                            &r_f_eq(kd),r_f_eq(kd+1),excl_last=.true.)
                        chckerr('')
                    end do
                    r_f_x(size(r_f_x)) = r_f_eq(size(r_f_eq))

                    ! set up normal limits
                    x_limits(1) = eq_limits(1)+sum(div(1:eq_limits(1)-1))
                    x_limits(2) = eq_limits(2)+sum(div(1:eq_limits(2)-1))

                    ! plot
```

```
                    if (rank.eq.0) then
                        allocate(r_f_plot(size(r_f_x),4))
                        r_f_plot(1:size(r_f_eq),1) = r_f_eq
                        r_f_plot(1:size(r_f_eq),3) = [(kd,kd=1,size(r_f_eq))]
                        r_f_plot(size(r_f_eq)+1:size(r_f_x),1) = &
                            &r_f_eq(size(r_f_eq))
                        r_f_plot(size(r_f_eq)+1:size(r_f_x),3) = size(r_f_eq)
                        r_f_plot(:,2) = r_f_x
                        r_f_plot(:,4) = [(kd,kd=1,size(r_f_x))]
                        r_f_plot(:,1:2) = r_f_plot(:,1:2)/max_flux_f*2*pi
                        call print_ex_2d(['eq','X '],'r_F_X',&
                            &r_f_plot(:,3:4),x=r_f_plot(:,1:2),draw=.false.)
                        call draw_ex(['eq','X '],'r_F_X',2,1,&
                            &.false.)
                    end if

                    ! output
                    call writo('Enriching the equilibrium grid to have jq &
                        &change not more than '//trim(r2strt(max_njq_change)))
                    call lvl_ud(1)
                    if (any(div.gt.0)) then
                        call writo('Added '//trim(i2str(sum(div)))//' points')
                    else
                        call writo('But it was not necessary to  add points')
                    end if
                    call lvl_ud(-1)
                end select
            end function calc_norm_range_pb3d_x

            !> \public PB3D solution version
            !!
            !! The normal range is determined by simply dividing the solution range,
            !! a ghost range is required.
            !!
            !! By default, this routine uses  \c sol_n_procs processes, but this can
            !! be overruled.
            integer function calc_norm_range_pb3d_sol(sol_limits,r_F_sol,n_procs) &
                &result(ierr)                                           ! PB3D version for solution grid
                use num_vars, only: sol_n_procs, rank, norm_disc_prec_sol
                use eq_vars, only: max_flux_f
                use x_vars, only: min_r_sol, max_r_sol
                use num_utilities, only: round_with_tol
                use grid_vars, only: n_r_sol

                character(*), parameter :: rout_name = 'calc_norm_range_PB3D_sol'

                ! input / output
                integer, intent(inout) :: sol_limits(2)                 !< min. and max. index of sol grid for this process
                real(dp), intent(inout) :: r_f_sol(:)                   !< solution r_F
                integer, intent(in), optional :: n_procs                !< how many processes used

                ! local variables
                integer :: id                                           ! counter
                integer, allocatable :: loc_n_r_sol(:)                  ! local nr. of normal points in solution grid
                integer :: n_procs_loc                                  ! local n_procs

                ! initialize ierr
                ierr = 0

                ! set local n_procs
                n_procs_loc = sol_n_procs
                if (present(n_procs)) n_procs_loc = n_procs

                ! set loc_n_r_sol
                allocate(loc_n_r_sol(n_procs_loc))
                loc_n_r_sol = n_r_sol/n_procs_loc                       ! number of radial points on this processor
                loc_n_r_sol(1:mod(n_r_sol,n_procs_loc)) = &
                    &loc_n_r_sol(1:mod(n_r_sol,n_procs_loc)) + 1        ! add a mode to if there is a remainder

                ! set sol_limits
                if (rank.lt.n_procs_loc) then
                    sol_limits = &
                        &[sum(loc_n_r_sol(1:rank))+1,sum(loc_n_r_sol(1:rank+1))]

                    ! ghost regions of width 2*norm_disc_prec_sol
                    sol_limits(1) = max(sol_limits(1)-norm_disc_prec_sol,1)
                    sol_limits(2) = min(sol_limits(2)+norm_disc_prec_sol,n_r_sol)
                else
                    sol_limits(1) = n_r_sol
                    sol_limits(2) = n_r_sol
                end if

                ! calculate r_F_sol in range from min_r_sol to max_r_sol
                r_f_sol = [(min_r_sol + (id-1.0_dp)/(n_r_sol-1.0_dp)*&
                    &(max_r_sol-min_r_sol),id=1,n_r_sol)]
```

```
            ! round with standard tolerance
            ierr = round_with_tol(r_f_sol,0.0_dp,1.0_dp)
            chckerr('')

            ! translate to the real normal variable in range from
            ! 0..flux/2pi
            r_f_sol = r_f_sol*max_flux_f/(2*pi)
        end function calc_norm_range_pb3d_sol

        !> \public POST version
        !!
        !! The normal range is determined by simply dividing a possible subset
        !! of the solution range, indicated by \c min_r_plot and \c max_r_sol,
        !! including a ghost range and getting a bounding equilibrium and
        !! perturbation range.
        subroutine calc_norm_range_post(eq_limits,X_limits,sol_limits,r_F_eq,&
            &r_F_X,r_F_sol)                                                 ! POST version

            use num_vars, only: n_procs, rank, norm_disc_prec_sol, &
                &min_r_plot, max_r_plot, x_grid_style
            use eq_vars, only: max_flux_f
            use grid_utilities, only: find_compr_range

            ! input / output
            integer, intent(inout) :: eq_limits(2)                          !< min. and max. index of eq grid for this process
            integer, intent(inout) :: X_limits(2)                           !< min. and max. index of X grid for this process
            integer, intent(inout) :: sol_limits(2)                         !< min. and max. index of sol grid for this process
            real(dp), intent(in) :: r_F_eq(:)                               !< eq r_F
            real(dp), intent(in) :: r_F_X(:)                                !< X r_F
            real(dp), intent(in) :: r_F_sol(:)                              !< sol r_F

            ! local variables
            integer :: sol_limits_tot(2)                                    ! total solution limits
            integer :: n_r_sol                                              ! total nr. of normal points in solution grid
            integer, allocatable :: loc_n_r_sol(:)                          ! local nr. of normal points in solution grid
            real(dp) :: min_sol, max_sol                                    ! min. and max. of r_F_sol in range of this process

            ! initialize ierr
            ierr = 0

            ! get min and max of solution range
            min_sol = max(minval(r_f_sol),min_r_plot*max_flux_f/(2*pi))
            max_sol = min(maxval(r_f_sol),max_r_plot*max_flux_f/(2*pi))

            ! find the solution index that comprises this range
            call find_compr_range(r_f_sol,[min_sol,max_sol],sol_limits_tot)

            ! initialize n_r_sol
            n_r_sol = sol_limits_tot(2)-sol_limits_tot(1)+1
            allocate(loc_n_r_sol(n_procs))

            ! divide the solution grid equally over all the processes
            loc_n_r_sol = n_r_sol/n_procs                                   ! number of radial points on this processor
            loc_n_r_sol(1:mod(n_r_sol,n_procs)) = &
                &loc_n_r_sol(1:mod(n_r_sol,n_procs)) + 1                    ! add a mode to if there is a remainder

            ! set sol_limits
            sol_limits = [sum(loc_n_r_sol(1:rank))+1,sum(loc_n_r_sol(1:rank+1))]
            sol_limits = sol_limits + sol_limits_tot(1) - 1
            if (rank.gt.0) sol_limits(1) = sol_limits(1)-norm_disc_prec_sol ! ghost region for num. deriv.
            if (rank.lt.n_procs-1) sol_limits(2) = &
                &sol_limits(2)+norm_disc_prec_sol+1                         ! ghost region for num. deriv. and overlap for int_vol
            min_sol = minval(r_f_sol(sol_limits(1):sol_limits(2)))
            max_sol = maxval(r_f_sol(sol_limits(1):sol_limits(2)))

            ! determine eq_limits: smallest eq and X range comprising sol range
            call find_compr_range(r_f_eq,[min_sol,max_sol],eq_limits)
            select case (x_grid_style)
                case (1,3)                                                  ! equilibrium or enriched
                    call find_compr_range(r_f_x,[min_sol,max_sol],x_limits)
                case (2)                                                    ! solution
                    x_limits = sol_limits
            end select
        end subroutine calc_norm_range_post
    end function calc_norm_range

    !> Sets up the equilibrium grid.
    !!
    !! The following variables are calculated:
```

```
    !!  - equilibrium variables (eq)
    !!  - perturbation variables (X)
    !!
    !! For the  implementation of the  equilibrium grid  the normal part  of the
    !! grid  is always  given by  the output  of the  equilibrium code,  but the
    !! angular part depends on the style:
    !!  -  VMEC:  The  output  is   analytic  in  the  angular  coordinates,  so
    !!  field-aligned coordinates are used.  Later, in calc_ang_grid_eq_b(), the
    !!  angular variables are calculated.
    !!  - HELENA: The output is given on a poloidal grid (no toroidal dependency
    !!  due to axisymmetry), which is used.
    !!
    !! \return ierr
    integer function setup_grid_eq(grid_eq,eq_limits) result(ierr)
        use num_vars, only: eq_style, eq_jobs_lims, eq_job_nr
        use grid_vars, only: n_r_eq, n_alpha
        use helena_vars, only: nchi, chi_h
        use rich_vars, only: n_par_x

        character(*), parameter :: rout_name = 'setup_grid_eq'

        ! input / output
        type(grid_type), intent(inout) :: grid_eq                              !< equilibrium grid
        integer, intent(in) :: eq_limits(2)                                    !< min. and max. index of eq grid of this process

        ! local variables
        integer :: id                                                          ! counter

        ! initialize ierr
        ierr = 0

        ! set up general equilibrium grid:
        ! choose which equilibrium style is being used:
        !   1:  VMEC
        !   2:  HELENA
        select case (eq_style)
            case (1)                                                           ! VMEC
                ! user output
                call writo('Field-aligned with '//trim(i2str(n_r_eq))//&
                    &' normal and '//trim(i2str(n_par_x))//' parallel points')

                ! create grid with eq_jobs_lims
                ierr = grid_eq%init([eq_jobs_lims(2,eq_job_nr)-&
                    &eq_jobs_lims(1,eq_job_nr)+1,n_alpha,n_r_eq],eq_limits)
                chckerr('')
            case (2)                                                           ! HELENA
                ! user output
                call writo('Identical to the equilibrium input grid')
                call writo(trim(i2str(n_r_eq))//' normal and '//&
                    &trim(i2str(nchi))//' angular points')
                call writo('Poloidal range '//trim(r2strt(chi_h(1)))//'..'//&
                    &trim(r2strt(chi_h(nchi))))
                call writo('Will be interpolated to a field-aligned grid &
                    &later')

                ! create grid
                ierr = grid_eq%init([nchi,1,n_r_eq],eq_limits)                 ! axisymmetric equilibrium
                chckerr('')

                ! copy angular grid from HELENA
                do id = 1,grid_eq%n(1)
                    grid_eq%theta_E(id,:,:) = chi_h(id)
                end do
                grid_eq%zeta_E = 0._dp

                ! convert to Flux coordinates (trivial)
                grid_eq%theta_F = grid_eq%theta_E
                grid_eq%zeta_F = grid_eq%zeta_E
        end select
    end function setup_grid_eq

    !> Sets up the field-aligned equilibrium grid.
    !!
    !! This serves  as a bridge  to the solution grid,  as it contains  the same
    !! normal coordinate  as the general  grid, but the angular  coordinates are
    !! defined by the solution grid.
    !!
    !! Optionally, only  half the  grid can  be calculated  (i.e. only  the even
    !! points), which is used for Richardson levels greater than 1.
    !!
    !! \note In contrast  to \c setup_grid_eq, the angular  coordinates are also
    !! calculated here.
    !!
    !! \return ierr
    integer function setup_grid_eq_b(grid_eq,grid_eq_B,eq,only_half_grid) &
        &result(ierr)
```

```
        use num_vars, only: eq_style, eq_jobs_lims, eq_job_nr
        use grid_vars, only: n_alpha

        character(*), parameter :: rout_name = 'setup_grid_eq_B'

        ! input / output
        type(grid_type), intent(inout) :: grid_eq                               !< general equilibrium grid
        type(grid_type), intent(inout) :: grid_eq_b                             !< field-aligned equilibrium grid
        type(eq_1_type), intent(in) :: eq                                       !< flux equilibrium variables
        logical, intent(in), optional :: only_half_grid                         !< calculate only half grid with even points

        ! local variables
        character(len=max_str_ln) :: err_msg                                    ! error message

        ! initialize ierr
        ierr = 0

        ! choose which equilibrium style is being used:
        !   1:  VMEC
        !   2:  HELENA
        select case (eq_style)
            case (1)                                                            ! VMEC
                ! the grid is already field aligned
                ierr = 1
                err_msg = 'The grid is already field-aligned for VMEC'
                chckerr(err_msg)
            case (2)                                                            ! HELENA
                ! create grid with eq_jobs_lims
                ierr = grid_eq_b%init([eq_jobs_lims(2,eq_job_nr)-&
                    &eq_jobs_lims(1,eq_job_nr)+1,n_alpha,grid_eq%n(3)],&
                    &[grid_eq%i_min,grid_eq%i_max])                             ! only one field line
                chckerr('')

                ! copy the normal coords.
                grid_eq_b%loc_r_E = grid_eq%loc_r_E
                grid_eq_b%loc_r_F = grid_eq%loc_r_F
                grid_eq_b%r_E = grid_eq%r_E
                grid_eq_b%r_F = grid_eq%r_F

                ! calculate the angular grid that follows the magnetic field
                ierr = calc_ang_grid_eq_b(grid_eq_b,eq,only_half_grid)
                chckerr('')
        end select
    end function setup_grid_eq_b

    !> Sets  up  the  general  perturbation  grid,  in  which  the  perturbation
    !! variables are calculated.
    !!
    !! For \c  X_grid_style 1, this grid  is identical to the  equilibrium grid,
    !! and  for \c  X_grid_style  2,it  has the  same  angular  extent but  with
    !! different normal points, indicated by the variable \c r_F_X.
    !!
    !! \return ierr
    integer function setup_grid_x(grid_eq,grid_X,r_F_X,X_limits) result(ierr)
        use num_vars, only: norm_disc_prec_x, x_grid_style
        use grid_utilities, only: coord_f2e
        use num_utilities, only: spline

        character(*), parameter :: rout_name = 'setup_grid_X'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(grid_type), intent(inout) :: grid_x                                !< perturbation grid
        real(dp), intent(in), allocatable :: r_f_x(:)                           !< points of perturbation grid
        integer, intent(in) :: x_limits(2)                                      !< min. and max. index of perturbation grid
    of this process

        ! local variables
        integer :: id, jd                                                       ! counters

        ! initialize ierr
        ierr = 0

        select case (x_grid_style)
            case (1)                                                            ! equilibrium
                ! X grid identical to equilibrium grid
                ierr = grid_eq%copy(grid_x)
                chckerr('')
            case (2,3)                                                          ! solution and enriched
                ! create grid
                ierr = grid_x%init([grid_eq%n(1:2),size(r_f_x)],x_limits)
                chckerr('')

                ! set Flux variables
                grid_x%r_F = r_f_x
                grid_x%loc_r_F = r_f_x(x_limits(1):x_limits(2))

```

```
                ! convert to Equilibrium variables
                ierr = coord_f2e(grid_eq,grid_x%r_F,grid_x%r_E,&
                    &r_f_array=grid_eq%r_F,r_e_array=grid_eq%r_E)
                chckerr('')
                ierr = coord_f2e(grid_eq,grid_x%loc_r_F,grid_x%loc_r_E,&
                    &r_f_array=grid_eq%r_F,r_e_array=grid_eq%r_E)
                chckerr('')

                ! interpolate
                do jd = 1,grid_eq%n(2)
                    do id = 1,grid_eq%n(1)
                        ierr = spline(grid_eq%loc_r_F,grid_eq%theta_E(id,jd,:),&
                            &grid_x%loc_r_F,grid_x%theta_E(id,jd,:),&
                            &ord=norm_disc_prec_x)
                        chckerr('')
                        ierr = spline(grid_eq%loc_r_F,grid_eq%zeta_E(id,jd,:),&
                            &grid_x%loc_r_F,grid_x%zeta_E(id,jd,:),&
                            &ord=norm_disc_prec_x)
                        chckerr('')
                        ierr = spline(grid_eq%loc_r_F,grid_eq%theta_F(id,jd,:),&
                            &grid_x%loc_r_F,grid_x%theta_F(id,jd,:),&
                            &ord=norm_disc_prec_x)
                        chckerr('')
                        ierr = spline(grid_eq%loc_r_F,grid_eq%zeta_F(id,jd,:),&
                            &grid_x%loc_r_F,grid_x%zeta_F(id,jd,:),&
                            &ord=norm_disc_prec_x)
                        chckerr('')
                    end do
                end do
            end select
        end function setup_grid_x

        !> Sets up  the general solution grid,  in which the solution  variables are
        !! calculated.
        !!
        !! For the  solution grid,  only one  parallel point  is used,  but possibly
        !! multiple geodesic points, equal to the number of field lines, \c n_alpha.
        !!
        !! \return ierr
        integer function setup_grid_sol(grid_eq,grid_X,grid_sol,r_F_sol,&
            &sol_limits) result(ierr)

            use num_vars, only: n_procs, x_grid_style
            use grid_utilities, only: coord_f2e

            character(*), parameter :: rout_name = 'setup_grid_sol'

            ! input / output
            type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
            type(grid_type), intent(in) :: grid_x                                   !< perturbation grid
            type(grid_type), intent(inout) :: grid_sol                              !< solution grid
            real(dp), intent(in) :: r_f_sol(:)                                      !< points of solution grid
            integer, intent(in) :: sol_limits(2)                                    !< min. and max. index of sol grid of this process

            ! initialize ierr
            ierr = 0

            ! output
            call writo(trim(i2str(size(r_f_sol)))//' normal  points')

            select case (x_grid_style)
                case (1,3)                                                          ! equilibrium or enriched
                    ! create grid
                    ierr = grid_sol%init([0,0,size(r_f_sol)],sol_limits,&
                        &divided=n_procs.gt.1)
                    chckerr('')

                    ! set Flux variables
                    grid_sol%r_F = r_f_sol
                    grid_sol%loc_r_F = r_f_sol(sol_limits(1):sol_limits(2))

                    ! convert to Equilibrium variables
                    ierr = coord_f2e(grid_eq,grid_sol%r_F,grid_sol%r_E,&
                        &r_f_array=grid_eq%r_F,r_e_array=grid_eq%r_E)
                    chckerr('')
                    ierr = coord_f2e(grid_eq,grid_sol%loc_r_F,grid_sol%loc_r_E,&
                        &r_f_array=grid_eq%r_F,r_e_array=grid_eq%r_E)
                    chckerr('')
                case (2)                                                            ! solution
                    ! solution grid  identical to perturation  grid but with  only 1
                    ! parallel point.
                    ierr = grid_sol%init([0,0,grid_x%n(3)],&
                        &[grid_x%i_min,grid_x%i_max],grid_x%divided)
                    chckerr('')
                    grid_sol%r_F = grid_x%r_F
                    grid_sol%loc_r_F = grid_x%r_F(sol_limits(1):sol_limits(2))
```

```
                grid_sol%r_E = grid_x%r_E
                grid_sol%loc_r_E = grid_x%r_E(sol_limits(1):sol_limits(2))
        end select
    end function setup_grid_sol

    !> Calculate equilibrium grid that follows magnetic field lines.
    !!
    !! This grid is  different from the equilibrium grid  from setup_grid_eq for
    !! HELENA, as the  latter is the output grid from  HELENA, which is situated
    !! in a single  poloidal cross-section, as opposed to  a really field-aligne
    !! grid.
    !!
    !! For VMEC, this is not used as the grid is field-aligned from the start.
    !!
    !! \note
    !!  -# The end-points are included for  the grids in the parallel direction.
    !!  This is to facilitate working with the trapezoidal rule or Simpson's 3/8
    !!  rule for integration. This is \b NOT valid in general!
    !!  -# by  setting the flag \c  only_half_grid, only the even  points of the
    !!  parallel  grid are  calculated, which  is useful  for higher  Richardson
    !!  levels with VMEC so that only  new angular points are calculated and the
    !!  old ones reused.
    !!
    !! \return ierr
    integer function calc_ang_grid_eq_b(grid_eq,eq,only_half_grid) result(ierr)
        use num_vars, only: use_pol_flux_f, use_pol_flux_e, &
            &eq_style, tol_zero, eq_job_nr, eq_jobs_lims
        use grid_vars, only: min_par_x, max_par_x, alpha, n_alpha
        use eq_vars, only: max_flux_e
        use grid_utilities, only: coord_f2e, calc_eqd_grid, calc_n_par_x_rich
        use eq_utilities, only: print_info_eq
        use rich_vars, only: n_par_x
        use x_vars, only: min_r_sol, max_r_sol
#if ldebug
        use grid_utilities, only: coord_e2f
#endif

        character(*), parameter :: rout_name = 'calc_ang_grid_eq_B'

        ! input / output
        type(grid_type), intent(inout) :: grid_eq                                !< equilibrium grid of which to calculate angular part
        type(eq_1_type), intent(in), target :: eq                                !< flux equilibrium variables
        logical, intent(in), optional :: only_half_grid                          !< calculate only half grid with even points

        ! local variables
        character(len=max_str_ln) :: err_msg                                     ! error message
        real(dp), allocatable :: r_e_loc(:)                                      ! flux in Equilibrium coords.
        real(dp), pointer :: flux_f(:) => null()                                 ! flux that the F uses as normal coord.
        real(dp), pointer :: flux_e(:) => null()                                 ! flux that the E uses as normal coord.
        real(dp) :: r_f_factor, r_e_factor                                       ! mult. factors for r_F and r_E
        integer :: pmone                                                         ! plus or minus one
        integer :: id, jd, kd                                                    ! counters
        integer :: n_par_x_loc                                                   ! local n_par_X
        logical :: only_half_grid_loc                                            ! local only_half_grid
        real(dp), allocatable :: theta_f_loc(:,:,:)                              ! local theta_F
        real(dp), allocatable :: zeta_f_loc(:,:,:)                               ! local zeta_F

        ! initialize ierr
        ierr = 0

        call lvl_ud(1)

        ! set local only_half_grid
        only_half_grid_loc = .false.
        if (present(only_half_grid)) only_half_grid_loc = only_half_grid

        ! set local n_par_X_rich
        ierr = calc_n_par_x_rich(n_par_x_loc,only_half_grid_loc)
        chckerr('')

        ! user output
        call writo('for '//trim(i2str(grid_eq%n(3)))//&
            &' values on normal range '//trim(r2strt(min_r_sol))//'..'//&
            &trim(r2strt(max_r_sol)))
        call writo('for '//trim(i2str(n_par_x))//' values on parallel &
            &range '//trim(r2strt(min_par_x))//'pi..'//&
            &trim(r2strt(max_par_x))//'pi')
        call lvl_ud(1)
        if (only_half_grid_loc) call writo('for this Richardson level, only &
            &the even points are setup up')
        call print_info_eq(n_par_x_loc)
        call lvl_ud(-1)

        ! set up flux_E and plus minus one
        ! Note: this routine  is similar to calc_loc_r, but  that routine cannot
        ! be used here because it is possible that the FD quantities are not yet
```

```
        ! defined.
        ! choose which equilibrium style is being used:
        !   1:  VMEC
        !   2:  HELENA
        select case (eq_style)
            case (1)                                                    ! VMEC
                if (use_pol_flux_e) then                                ! normalized poloidal flux
                    flux_e => eq%flux_p_E(:,0)
                else                                                    ! normalized toroidal flux
                    flux_e => eq%flux_t_E(:,0)
                end if
                r_e_factor = max_flux_e
                pmone = -1                                              ! conversion VMEC LH -> RH coord. system
            case (2)                                                    ! HELENA
                flux_e => eq%flux_p_E(:,0)
                r_e_factor = 2*pi
                pmone = 1
        end select

        ! set up flux_F
        if (use_pol_flux_f) then                                        ! poloidal flux / 2pi
            flux_f => eq%flux_p_E(:,0)
            r_f_factor = 2*pi
        else                                                            ! toroidal flux / 2pi
            flux_f => eq%flux_t_E(:,0)
            r_f_factor = pmone*2*pi                                     ! possible conversion VMEC LH -> RH coord.
    system
        end if

        ! set up parallel angle in  Flux coordinates on equidistant grid
        ! and use this to calculate the other angle as well
        allocate(theta_f_loc(n_par_x,n_alpha,grid_eq%loc_n_r))
        allocate(zeta_f_loc(n_par_x,n_alpha,grid_eq%loc_n_r))
        if (use_pol_flux_f) then                                        ! parallel angle theta
            ierr = calc_eqd_grid(theta_f_loc,min_par_x*pi,&
                &max_par_x*pi,1,excl_last=.false.)                      ! first index corresponds to parallel angle
            chckerr('')
            do kd = 1,grid_eq%loc_n_r
                zeta_f_loc(:,:,kd) = pmone*eq%q_saf_E(kd,0)*&
                    &theta_f_loc(:,:,kd)
            end do
            do jd = 1,n_alpha
                zeta_f_loc(:,jd,:) = zeta_f_loc(:,jd,:) + alpha(jd)
            end do
        else                                                            ! parallel angle zeta
            ierr = calc_eqd_grid(zeta_f_loc,min_par_x*pi,&
                &max_par_x*pi,1,excl_last=.false.)                      ! first index corresponds to parallel angle
            chckerr('')
            do kd = 1,grid_eq%loc_n_r
                theta_f_loc(:,:,kd) = pmone*eq%rot_t_E(kd,0)*&
                    &zeta_f_loc(:,:,kd)
            end do
            do jd = 1,n_alpha
                theta_f_loc(:,jd,:) = theta_f_loc(:,jd,:) - alpha(jd)
            end do
        end if

        ! set up grid_eq angular F coordinates
        if (only_half_grid_loc) then
            do id = eq_jobs_lims(1,eq_job_nr),eq_jobs_lims(2,eq_job_nr)
                grid_eq%theta_F(id-eq_jobs_lims(1,eq_job_nr)+1,:,:) = &
                    &theta_f_loc(2*id,:,:)
                grid_eq%zeta_F(id-eq_jobs_lims(1,eq_job_nr)+1,:,:) = &
                    &zeta_f_loc(2*id,:,:)
            end do
        else
            grid_eq%theta_F = theta_f_loc(eq_jobs_lims(1,eq_job_nr):&
                &eq_jobs_lims(2,eq_job_nr),:,:)
            grid_eq%zeta_F = zeta_f_loc(eq_jobs_lims(1,eq_job_nr):&
                &eq_jobs_lims(2,eq_job_nr),:,:)
        end if

        ! allocate local r_E
        allocate(r_e_loc(size(flux_f)))

        ! convert  Flux  coordinates  to  Equilibrium  coordinates  (use
        ! custom flux_E and  flux_F because the Flux  quantities are not
        ! yet calculated)
        call writo('convert F to E coordinates')
        call lvl_ud(1)
        ierr = coord_f2e(grid_eq,grid_eq%loc_r_F,grid_eq%theta_F,&
            &grid_eq%zeta_F,r_e_loc,grid_eq%theta_E,grid_eq%zeta_E,&
            &r_f_array=flux_f/r_f_factor,r_e_array=flux_e/r_e_factor)
        chckerr('')
        call lvl_ud(-1)

        ! test whether r_E_loc indeed corresponds to loc_r_E of eq grid
```

```
        if (maxval(abs(grid_eq%loc_r_E-r_e_loc)).gt.10*tol_zero) then
            ierr = 1
            err_msg = 'loc_r_E of equilibrium grid is not recovered'
            chckerr(err_msg)
        end if

#if ldebug
        if (debug_calc_ang_grid_eq_b) then
            ! test whether F variables recovered
            deallocate(theta_f_loc,zeta_f_loc)
            allocate(theta_f_loc(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
            allocate(zeta_f_loc(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
            ierr = coord_e2f(grid_eq,grid_eq%loc_r_E,grid_eq%theta_E,&
                &grid_eq%zeta_E,grid_eq%loc_r_F,theta_f_loc,zeta_f_loc,&
                &r_e_array=flux_e/r_e_factor,r_f_array=flux_f/r_f_factor)
            chckerr('')

            ! plot difference
            call plot_diff_hdf5(grid_eq%theta_F,theta_f_loc,'TEST_theta_F',&
                &grid_eq%n,[0,0,grid_eq%i_min-1],'test whether F variable are &
                &recovered',output_message=.true.)
            call plot_diff_hdf5(grid_eq%zeta_F,zeta_f_loc,'TEST_zeta_F',&
                &grid_eq%n,[0,0,grid_eq%i_min-1],'test whether F variable are &
                &recovered',output_message=.true.)
        end if
#endif

        ! deallocate local variables
        deallocate(r_e_loc)
        nullify(flux_f,flux_e)

        call lvl_ud(-1)
    end function calc_ang_grid_eq_b

    !> Redistribute a grid to match the normal distribution of solution grid.
    !!
    !! The routine first calculates the smallest eq range that comprises the sol
    !! range.  Then, it  gets the  lowest equilibrium  limits able  to setup  an
    !! output grid that starts at index 1. After determining the output grid, it
    !! then sends the variables to their new processes using MPI.
    !!
    !! \note
    !!  -# Only the  Flux variables are saved.
    !!  -# the redistributed grid has trimmed  outer limits, i.e. it starts at 1
    !!  and ends at the upper limit of  the last process. This can be turned off
    !!  optionally using \c no_outer_trim.
    !!
    !! \return ierr
    integer function redistribute_output_grid(grid,grid_out,no_outer_trim) &
        &result(ierr)
        use grid_utilities, only: find_compr_range
        use mpi_utilities, only: redistribute_var, get_ser_var
        use grid_vars, only: n_r_sol
        use num_vars, only: n_procs

        character(*), parameter :: rout_name = 'redistribute_output_grid'

        ! input / output
        type(grid_type), intent(in) :: grid                                     !< equilibrium grid variables
        type(grid_type), intent(inout) :: grid_out                              !< redistributed equilibrium grid variables
        logical, intent(in), optional :: no_outer_trim                          !< do not trim the outer limits

        ! local variables
        integer :: id                                                           ! counter
        integer :: eq_limits(2)                                                 ! normal limits for equilibrium variables
        integer :: sol_limits(2)                                                ! normal limits for perturbation variables
        integer :: n_out(3)                                                     ! n of grid_out
        integer :: i_lim_tot(2)                                                 ! total limits of grid
        integer :: i_lim_out(2)                                                 ! limits of grid_out
        integer :: lims(2), lims_dis(2)                                         ! limits and distributed limits, taking into
    account the angular extent
        real(dp), allocatable :: r_f_sol(:)                                     ! perturbation r_F
        real(dp), allocatable :: temp_var(:)                                    ! temporary variable
        integer, allocatable :: temp_lim(:)                                     ! temporary limit
        integer, allocatable :: eq_limits_tot(:,:)                              ! total equilibrium limits
        logical :: no_outer_trim_loc                                            ! local no_outer_trim

        ! initialize ierr
        ierr = 0

        ! calculate normal range for solution and save in perturbation variables
        allocate(r_f_sol(n_r_sol))
        ierr = calc_norm_range('PB3D_sol',sol_limits=sol_limits,r_f_sol=r_f_sol)
        chckerr('')

        ! determine eq_limits: smallest eq range comprising X range
        call find_compr_range(grid%r_F,r_f_sol(sol_limits),eq_limits)
```

```fortran

        ! get lowest equilibrium limits to be  able to setup an output grid that
        ! starts at index 1
        allocate(eq_limits_tot(2,n_procs))
        do id = 1,2
            ierr = get_ser_var(eq_limits(id:id),temp_lim,scatter=.true.)
            chckerr('')
            eq_limits_tot(id,:) = temp_lim
        end do

        ! set up redistributed grid
        no_outer_trim_loc = .false.
        if (present(no_outer_trim)) no_outer_trim_loc = no_outer_trim
        if (no_outer_trim_loc) then
            n_out = grid%n
            i_lim_tot = [1,n_out(3)]
            i_lim_out = eq_limits
        else
            n_out(1:2) = grid%n(1:2)
            n_out(3) = eq_limits_tot(2,n_procs)-eq_limits_tot(1,1)+1
            i_lim_tot = [eq_limits_tot(1,1),eq_limits_tot(2,n_procs)]
            i_lim_out = eq_limits-eq_limits_tot(1,1)+1
        end if
        ierr = grid_out%init(n_out,i_lim_out)
        chckerr('')

        ! set up limits taking into account angular extent and temporary var
        lims(1) = product(grid%n(1:2))*(grid%i_min-1)+1
        lims(2) = product(grid%n(1:2))*grid%i_max
        lims_dis(1) = product(grid%n(1:2))*(eq_limits(1)-1)+1
        lims_dis(2) = product(grid%n(1:2))*eq_limits(2)
        allocate(temp_var(lims_dis(2)-lims_dis(1)+1))

        ! copy total variables
        ! r_F
        grid_out%r_F = grid%r_F(i_lim_tot(1):i_lim_tot(2))
        ! r_E
        grid_out%r_E = grid%r_E(i_lim_tot(1):i_lim_tot(2))

        ! distribute local variables
        ! theta_F
        ierr = redistribute_var(reshape(grid%theta_F,[size(grid%theta_F)]),&
            &temp_var,lims,lims_dis)
        chckerr('')
        grid_out%theta_F = reshape(temp_var,shape(grid_out%theta_F))
        ! zeta_F
        ierr = redistribute_var(reshape(grid%zeta_F,[size(grid%zeta_F)]),&
            &temp_var,lims,lims_dis)
        chckerr('')
        grid_out%zeta_F = reshape(temp_var,shape(grid_out%zeta_F))
        ! theta_E
        ierr = redistribute_var(reshape(grid%theta_E,[size(grid%theta_E)]),&
            &temp_var,lims,lims_dis)
        chckerr('')
        grid_out%theta_E = reshape(temp_var,shape(grid_out%theta_E))
        ! zeta_E
        ierr = redistribute_var(reshape(grid%zeta_E,[size(grid%zeta_E)]),&
            &temp_var,lims,lims_dis)
        chckerr('')
        grid_out%zeta_E = reshape(temp_var,shape(grid_out%zeta_E))
        ! loc_r_F and loc_R_E
        if (grid_out%divided) then
            grid_out%loc_r_F = grid_out%r_F(grid_out%i_min:grid_out%i_max)
            grid_out%loc_r_E = grid_out%r_E(grid_out%i_min:grid_out%i_max)
        end if
    end function redistribute_output_grid

    !> Plots the grid in real 3-D space.
    !!
    !! This creates an animation that can be used by ParaView or VisIt.
    !!
    !! The equilibrium grid should contain the fieldline-oriented angles with \c
    !! ang_1 the parallel angle and \c ang_2 the field line label.
    !!
    !! \see See grid_vars.grid_type for a discussion on \c ang_1 and \c ang_2.
    !!
    !! \note
    !!  -# This procedure  does not use \c n_theta_plot and  \c n_zeta_plot from
    !!  num_vars, but instead temporarily overwrites them with its own, since it
    !!  is suposed to be 3-D also in the axisymmetric case.
    !!  -# The implementation is currently very slow.
    !!
    !! \return ierr
    integer function magn_grid_plot(grid) result(ierr)
        use num_vars, only: rank, no_plots, n_theta_plot, n_zeta_plot, &
            &eq_style, min_theta_plot, max_theta_plot, min_zeta_plot, &
            &max_zeta_plot
```

```
        use grid_utilities, only: trim_grid, extend_grid_f, calc_xyz_grid
        use vmec_utilities, only: calc_trigon_factors

        character(*), parameter :: rout_name = 'magn_grid_plot'

        ! input / output
        type(grid_type), intent(in) :: grid                                     !< fieldline-oriented equilibrium grid

        ! local variables
        real(dp), allocatable :: x_1(:,:,:), y_1(:,:,:), z_1(:,:,:)             ! X, Y and Z of surface in Axisymmetric coordinates
        real(dp), allocatable :: x_2(:,:,:), y_2(:,:,:), z_2(:,:,:)             ! X, Y and Z of magnetic field lines in Axisymmetric coordinates
        real(dp), pointer :: x_1_tot(:,:,:) => null()                           ! total X
        real(dp), pointer :: y_1_tot(:,:,:) => null()                           ! total Y
        real(dp), pointer :: z_1_tot(:,:,:) => null()                           ! total Z
        real(dp), pointer :: x_2_tot(:,:,:) => null()                           ! total X
        real(dp), pointer :: y_2_tot(:,:,:) => null()                           ! total Y
        real(dp), pointer :: z_2_tot(:,:,:) => null()                           ! total Z
        type(grid_type) :: grid_ext                                             ! angularly extended grid
        type(grid_type) :: grid_plot                                            ! grid for plotting
        integer :: n_theta_plot_old                                             ! backup of n_theta_plot
        integer :: n_zeta_plot_old                                              ! backup of n_zeta_plot
        real(dp) :: min_theta_plot_old, max_theta_plot_old                      ! backup of min and max_theta_plot
        real(dp) :: min_zeta_plot_old, max_zeta_plot_old                        ! backup of min and max_zeta_plot
        character(len=max_str_ln) :: anim_name                                  ! name of animation

        ! initialize ierr
        ierr = 0

        ! bypass plots if no_plots
        if (no_plots) return

        call writo('Plotting magnetic field and flux surfaces')

        call lvl_ud(1)

        ! 0. set up variables

        ! save n_theta_plot and n_zeta_plot and change them
        n_theta_plot_old = n_theta_plot
        n_zeta_plot_old = n_zeta_plot
        min_theta_plot_old = min_theta_plot
        max_theta_plot_old = max_theta_plot
        min_zeta_plot_old = min_zeta_plot
        max_zeta_plot_old = max_zeta_plot
        n_theta_plot = 40
        n_zeta_plot = 160
        min_theta_plot = 1
        max_theta_plot = 3
        min_zeta_plot = 0
        max_zeta_plot = 2

        ! extend grid
        ierr = extend_grid_f(grid,grid_ext,grid_eq=grid)
        chckerr('')

        ! restore n_theta_plot and n_zeta_plot
        n_theta_plot = n_theta_plot_old
        n_zeta_plot = n_zeta_plot_old
        min_theta_plot = min_theta_plot_old
        max_theta_plot = max_theta_plot_old
        min_zeta_plot = min_zeta_plot_old
        max_zeta_plot = max_zeta_plot_old

        ! trim extended grid into plot grid
        ierr = trim_grid(grid_ext,grid_plot)
        chckerr('')

        ! if VMEC, calculate trigonometric factors of plot grid
        if (eq_style.eq.1) then
            ierr = calc_trigon_factors(grid_plot%theta_E,grid_plot%zeta_E,&
                &grid_plot%trigon_factors)
            chckerr('')
        end if

        ! set animation name
        anim_name = 'Magnetic field in flux surfaces'

        ! 1. plot flux surfaces
        call writo('writing flux surfaces')

        ! calculate X_1,Y_1 and Z_1
        allocate(x_1(grid_plot%n(1),grid_plot%n(2),grid_plot%loc_n_r))
        allocate(y_1(grid_plot%n(1),grid_plot%n(2),grid_plot%loc_n_r))
        allocate(z_1(grid_plot%n(1),grid_plot%n(2),grid_plot%loc_n_r))
        ierr = calc_xyz_grid(grid,grid_plot,x_1,y_1,z_1)
```

```
        chckerr('')

        ! dealloc grid
        call grid_plot%dealloc()

        ! get pointers to full X, Y and Z
        ! The reason for this is that the plot  is not as simple as usual, so no
        ! divided  plots  are used,  and  also  efficiency  is not  the  biggest
        ! priority. Therefore,  all the plotting  of the  local is handled  by a
        ! single process, the master.
        call get_full_xyz(x_1,y_1,z_1,x_1_tot,y_1_tot,z_1_tot,'flux surfaces')

        ! 2. plot field lines
        call writo('writing field lines')

        ! trim grid into plot grid
        ierr = trim_grid(grid,grid_plot)
        chckerr('')

        ! if VMEC, calculate trigonometric factors of plot grid
        if (eq_style.eq.1) then
            ierr = calc_trigon_factors(grid_plot%theta_E,grid_plot%zeta_E,&
                &grid_plot%trigon_factors)
            chckerr('')
        end if

        ! calculate X_2,Y_2 and Z_2
        allocate(x_2(grid_plot%n(1),grid_plot%n(2),grid_plot%loc_n_r))
        allocate(y_2(grid_plot%n(1),grid_plot%n(2),grid_plot%loc_n_r))
        allocate(z_2(grid_plot%n(1),grid_plot%n(2),grid_plot%loc_n_r))
        ierr = calc_xyz_grid(grid,grid_plot,x_2,y_2,z_2)
        chckerr('')

        ! dealloc grids
        call grid_plot%dealloc()
        call grid_ext%dealloc()

        ! get pointers to full X, Y and Z
        ! The reason for this is that the plot  is not as simple as usual, so no
        ! divided  plots  are used,  and  also  efficiency  is not  the  biggest
        ! priority. Therefore,  all the plotting  of the  local is handled  by a
        ! single process, the master.
        call get_full_xyz(x_2,y_2,z_2,x_2_tot,y_2_tot,z_2_tot,'field lines')

        ierr = magn_grid_plot_hdf5(x_1_tot,x_2_tot,y_1_tot,y_2_tot,&
            &z_1_tot,z_2_tot,anim_name)
        chckerr('')

        ! clean up
        deallocate(x_1,y_1,z_1)
        deallocate(x_2,y_2,z_2)
        nullify(x_1_tot,y_1_tot,z_1_tot)
        nullify(x_2_tot,y_2_tot,z_2_tot)

        call lvl_ud(-1)

        call writo('Done plotting magnetic field and flux surfaces')
    contains
        ! get pointer to full plotting variables X, Y and Z
        !> \private
        subroutine get_full_xyz(X,Y,Z,X_tot,Y_tot,Z_tot,merge_name)
            use mpi_utilities, only: get_ser_var

            ! input / output
            real(dp), intent(in), target :: x(:,:,:), y(:,:,:), z(:,:,:)        ! X, Y and Z of either flux surfaces or magnetic field lines
            real(dp), intent(inout), pointer :: x_tot(:,:,:)                    ! pointer to full X
            real(dp), intent(inout), pointer :: y_tot(:,:,:)                    ! pointer to full Y
            real(dp), intent(inout), pointer :: z_tot(:,:,:)                    ! pointer to full Z
            character(len=*) :: merge_name                                      ! name of variable to be merged

            ! local variables
            real(dp), allocatable :: ser_xyz_loc(:)                             ! serial copy of XYZ_loc
            integer, allocatable :: tot_dim(:)                                  ! total dimensions for plot

            ! merge plots for flux surfaces if more than one process
            if (grid%divided) then                                              ! merge local plots
                ! user output
                call writo('Merging local plots for '//merge_name)

                ! get total dimension
                ierr = get_ser_var([size(x,3)],tot_dim)
                chckerr('')

                ! allocate total X, Y and Z (should have same sizes)
                if (rank.eq.0) then
                    allocate(x_tot(size(x,1),size(x,2),sum(tot_dim)))
```

```
                    allocate(y_tot(size(x,1),size(x,2),sum(tot_dim)))
                    allocate(z_tot(size(x,1),size(x,2),sum(tot_dim)))
                end if

                allocate(ser_xyz_loc(size(x(:,:,1))*sum(tot_dim)))
                ierr = get_ser_var(reshape(x,[size(x)]),ser_xyz_loc)
                chckerr('')
                if (rank.eq.0) x_tot = reshape(ser_xyz_loc,shape(x_tot))
                ierr = get_ser_var(reshape(y,[size(y)]),ser_xyz_loc)
                chckerr('')
                if (rank.eq.0) y_tot = reshape(ser_xyz_loc,shape(y_tot))
                ierr = get_ser_var(reshape(z,[size(z)]),ser_xyz_loc)
                chckerr('')
                if (rank.eq.0) z_tot = reshape(ser_xyz_loc,shape(z_tot))
            else                                                        ! just point
                x_tot => x
                y_tot => y
                z_tot => z
            end if
        end subroutine get_full_xyz

        ! Plot with HDF5
        !> \private
        integer function magn_grid_plot_hdf5(X_1,X_2,Y_1,Y_2,Z_1,Z_2,&
            &anim_name) result(ierr)
            use hdf5_ops, only: open_hdf5_file, add_hdf5_item, &
                &print_hdf5_top, print_hdf5_geom, print_hdf5_3d_data_item, &
                &print_hdf5_grid, close_hdf5_file
            use hdf5_vars, only: dealloc_xml_str, &
                & xml_str_type, hdf5_file_type
            use rich_vars, only: rich_lvl
            use grid_vars, only: n_alpha

            character(*), parameter :: rout_name = 'magn_grid_plot_HDF5'

            ! input / output
            real(dp), intent(in), pointer :: x_1(:,:,:), y_1(:,:,:), z_1(:,:,:) ! X, Y and Z of surface in Axisymmetric coordinates
            real(dp), intent(in), pointer :: x_2(:,:,:), y_2(:,:,:), z_2(:,:,:) ! X, Y and Z of magnetic field lines in Axisymmetric coordinates
            character(len=*), intent(in) :: anim_name                   ! name of animation

            ! local variables
            character(len=max_str_ln) :: plot_title(2)                  ! plot title for flux surface and for field lines
            character(len=max_str_ln) :: file_name                      ! name of file
            integer :: id, jd                                           ! counter
            type(hdf5_file_type) :: file_info                           ! file info
            type(xml_str_type), allocatable :: grids(:)                 ! grid in spatial collection (flux surface, field line)
            type(xml_str_type), allocatable :: space_col_grids(:)       ! grid with space collection at different times
            type(xml_str_type) :: time_col_grid                         ! grid with time collection
            type(xml_str_type) :: top                                   ! topology
            type(xml_str_type) :: xyz(3)                                ! data items for geometry
            type(xml_str_type) :: geom                                  ! geometry
            integer :: loc_dim(3,2)                                     ! local dimensions (flux surfaces, field lines)
            integer :: n_r                                              ! nr. of normal points

            ! initialize ierr
            ierr = 0

            ! only master
            if (rank.eq.0) then
                ! user output
                call writo('Drawing animation with HDF5')
                call lvl_ud(1)

                ! set up loc_dim and n_r
                loc_dim(:,1) = [size(x_1,1),size(x_1,2),1]
                loc_dim(:,2) = [size(x_2,1),1,1]
                n_r = size(x_1,3) - 1                                   ! should be same for all other X_i, Y_i and Z_i

                ! set up plot titles and file name
                plot_title(1) = 'Magnetic Flux Surfaces'
                plot_title(2) = 'Magnetic Field Lines'
                file_name = 'field_lines_in_flux_surfaces_R_'//&
                    &trim(i2str(rich_lvl))

                ! open HDF5 file
                ierr = open_hdf5_file(file_info,trim(file_name),&
                    &descr=anim_name,ind_plot=.true.)
                chckerr('')

                ! create grid  for flux surface and all field  lines and one for
```

```
            ! time collection
            allocate(grids(1+n_alpha))
            allocate(space_col_grids(n_r))

            ! loop over all normal points
            do id = 1,n_r
                ! A. start with flux surface

                ! print topology
                call print_hdf5_top(top,2,loc_dim(:,1))

                ! print data item for X
                ierr = print_hdf5_3d_data_item(xyz(1),file_info,&
                    &'X_surf_'//trim(i2str(id)),x_1(:,:,id:id),loc_dim(:,1))
                chckerr('')

                ! print data item for Y
                ierr = print_hdf5_3d_data_item(xyz(2),file_info,&
                    &'Y_surf_'//trim(i2str(id)),y_1(:,:,id:id),loc_dim(:,1))
                chckerr('')

                ! print data item for Z
                ierr = print_hdf5_3d_data_item(xyz(3),file_info,&
                    &'Z_surf_'//trim(i2str(id)),z_1(:,:,id:id),loc_dim(:,1))
                chckerr('')

                ! print geometry with X, Y and Z data item
                call print_hdf5_geom(geom,2,xyz,.true.)

                ! create a grid with the topology and the geometry
                ierr = print_hdf5_grid(grids(1),plot_title(1),1,&
                    &grid_top=top,grid_geom=geom,reset=.true.)
                chckerr('')

                ! B. end with magnetic field

                do jd = 1,n_alpha                                       ! iterate over all field lines
                    ! print topology
                    call print_hdf5_top(top,2,loc_dim(:,2))

                    ! print data item for X of this field line
                    ierr = print_hdf5_3d_data_item(xyz(1),file_info,&
                        &'X_field_'//trim(i2str(id))//'_'//trim(i2str(jd)),&
                        &x_2(:,jd:jd,id+1:id+1),loc_dim(:,2))
                    chckerr('')

                    ! print data item for Y of this field line
                    ierr = print_hdf5_3d_data_item(xyz(2),file_info,&
                        &'Y_field_'//trim(i2str(id))//'_'//trim(i2str(jd)),&
                        &y_2(:,jd:jd,id+1:id+1),loc_dim(:,2))
                    chckerr('')

                    ! print data item for Z of this field line
                    ierr = print_hdf5_3d_data_item(xyz(3),file_info,&
                        &'Z_field_'//trim(i2str(id))//'_'//trim(i2str(jd)),&
                        &z_2(:,jd:jd,id+1:id+1),loc_dim(:,2))
                    chckerr('')

                    ! print geometry with X, Y and Z data item
                    call print_hdf5_geom(geom,2,xyz,.true.)

                    ! create a grid with the topology and the geometry
                    ierr = print_hdf5_grid(grids(jd+1),plot_title(2),1,&
                        &grid_top=top,grid_geom=geom,reset=.true.)
                    chckerr('')
                end do

                ! C.  merge flux  surface and magnetic  field in  to spatial
                ! collection
                ierr = print_hdf5_grid(space_col_grids(id),&
                    &'spatial collection',3,grid_time=id*1._dp,&
                    &grid_grids=grids,reset=.true.)
                chckerr('')
            end do

            ! create grid collection from individual grids and reset them
            ierr = print_hdf5_grid(time_col_grid,'time collection',2,&
                &grid_grids=space_col_grids,reset=.true.)
            chckerr('')

            ! add collection grid to HDF5 file and reset it
            ierr = add_hdf5_item(file_info,time_col_grid,reset=.true.)
            chckerr('')

            ! close HDF5 file
            ierr = close_hdf5_file(file_info)
            chckerr('')
```

```
                call lvl_ud(-1)

                ! clean up
                do id = 1,2
                    call dealloc_xml_str(grids(id))
                end do
                call dealloc_xml_str(space_col_grids)
                call dealloc_xml_str(time_col_grid)
                call dealloc_xml_str(top)
                do id = 1,3
                    call dealloc_xml_str(xyz(id))
                end do
                call dealloc_xml_str(geom)
            end if
        end function magn_grid_plot_hdf5
    end function magn_grid_plot

    !> Print grid variables to an output file.
    !!
    !! If \c  rich_lvl is  provided, <tt>_R_[rich_lvl]</tt>  is appended  to the
    !! data name if it is > 0.
    !!
    !! Optionally, it  can be specified  that this  is a divided  parallel grid,
    !! corresponding to the variable \c eq_jobs_lims with index \c eq_job_nr. In
    !! this case,  the total grid  size is adjusted to  the one specified  by \c
    !! eq_jobs_lims and the grid is written as a subset.
    !!
    !! \note <tt>grid_</tt> is added in front the data_name.
    !!
    !! \return ierr
    integer function print_output_grid(grid,grid_name,data_name,rich_lvl,&
        &par_div,remove_previous_arrs) result(ierr)
        use num_vars, only: pb3d_name, eq_jobs_lims, eq_job_nr
        use hdf5_ops, only: print_hdf5_arrs
        use hdf5_vars, only: var_1d_type, &
            &max_dim_var_1d
        use grid_utilities, only: trim_grid

        character(*), parameter :: rout_name = 'print_output_grid'

        ! input / output
        type(grid_type), intent(in) :: grid                                     !< grid variables
        character(len=*), intent(in) :: grid_name                               !< name to display
        character(len=*), intent(in) :: data_name                               !< name under which to store
        integer, intent(in), optional :: rich_lvl                               !< Richardson level to reconstruct
        logical, intent(in), optional :: par_div                                !< is a parallely divided grid
        logical, intent(in), optional :: remove_previous_arrs                   !< remove previous variables if present

        ! local variables
        integer :: n_tot(3)                                                     ! total n
        integer :: par_id(2)                                                    ! local parallel interval
        type(grid_type) :: grid_trim                                            ! trimmed grid
        type(var_1d_type), allocatable, target :: grid_1d(:)                    ! 1D equivalent of grid variables
        type(var_1d_type), pointer :: grid_1d_loc => null()                     ! local element in grid_1D
        integer :: id                                                           ! counter
        logical :: par_div_loc                                                  ! local par_div

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Write '//trim(grid_name)//' grid variables to output &
            &file')
        call lvl_ud(1)

        ! trim grid
        ierr = trim_grid(grid,grid_trim)
        chckerr('')

        ! set local par_div
        par_div_loc = .false.
        if (present(par_div)) par_div_loc = par_div

        ! set total n and parallel interval
        n_tot = grid_trim%n
        par_id = [1,n_tot(1)]
        if (grid_trim%n(1).gt.0 .and. par_div_loc) then                         ! total grid includes all equilibrium jobs
            n_tot(1) = maxval(eq_jobs_lims)-minval(eq_jobs_lims)+1
            par_id = eq_jobs_lims(:,eq_job_nr)
        end if

        ! Set up the 1D equivalents of the perturbation variables
        allocate(grid_1d(max_dim_var_1d))

        ! set up variables grid_1D
        id = 1
```

```
        ! n
        grid_1d_loc => grid_1d(id); id = id+1
        grid_1d_loc%var_name = 'n'
        allocate(grid_1d_loc%tot_i_min(1),grid_1d_loc%tot_i_max(1))
        allocate(grid_1d_loc%loc_i_min(1),grid_1d_loc%loc_i_max(1))
        grid_1d_loc%tot_i_min = [1]
        grid_1d_loc%tot_i_max = [3]
        grid_1d_loc%loc_i_min = [1]
        grid_1d_loc%loc_i_max = [3]
        allocate(grid_1d_loc%p(3))
        grid_1d_loc%p = n_tot

        ! r_F
        grid_1d_loc => grid_1d(id); id = id+1
        grid_1d_loc%var_name = 'r_F'
        allocate(grid_1d_loc%tot_i_min(1),grid_1d_loc%tot_i_max(1))
        allocate(grid_1d_loc%loc_i_min(1),grid_1d_loc%loc_i_max(1))
        grid_1d_loc%tot_i_min = [1]
        grid_1d_loc%tot_i_max = [n_tot(3)]
        grid_1d_loc%loc_i_min = [grid_trim%i_min]
        grid_1d_loc%loc_i_max = [grid_trim%i_max]
        allocate(grid_1d_loc%p(size(grid_trim%loc_r_F)))
        grid_1d_loc%p = grid_trim%loc_r_F

        ! r_E
        grid_1d_loc => grid_1d(id); id = id+1
        grid_1d_loc%var_name = 'r_E'
        allocate(grid_1d_loc%tot_i_min(1),grid_1d_loc%tot_i_max(1))
        allocate(grid_1d_loc%loc_i_min(1),grid_1d_loc%loc_i_max(1))
        grid_1d_loc%tot_i_min = [1]
        grid_1d_loc%tot_i_max = [n_tot(3)]
        grid_1d_loc%loc_i_min = [grid_trim%i_min]
        grid_1d_loc%loc_i_max = [grid_trim%i_max]
        allocate(grid_1d_loc%p(size(grid_trim%loc_r_E)))
        grid_1d_loc%p = grid_trim%loc_r_E

        ! only for 3D grids
        if (product(grid%n(1:2)).ne.0) then
            ! theta_F
            grid_1d_loc => grid_1d(id); id = id+1
            grid_1d_loc%var_name = 'theta_F'
            allocate(grid_1d_loc%tot_i_min(3),grid_1d_loc%tot_i_max(3))
            allocate(grid_1d_loc%loc_i_min(3),grid_1d_loc%loc_i_max(3))
            grid_1d_loc%tot_i_min = [1,1,1]
            grid_1d_loc%tot_i_max = n_tot
            grid_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min]
            grid_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max]
            allocate(grid_1d_loc%p(size(grid_trim%theta_F)))
            grid_1d_loc%p = reshape(grid_trim%theta_F,[size(grid_trim%theta_F)])

            ! theta_E
            grid_1d_loc => grid_1d(id); id = id+1
            grid_1d_loc%var_name = 'theta_E'
            allocate(grid_1d_loc%tot_i_min(3),grid_1d_loc%tot_i_max(3))
            allocate(grid_1d_loc%loc_i_min(3),grid_1d_loc%loc_i_max(3))
            grid_1d_loc%tot_i_min = [1,1,1]
            grid_1d_loc%tot_i_max = n_tot
            grid_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min]
            grid_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max]
            allocate(grid_1d_loc%p(size(grid_trim%theta_E)))
            grid_1d_loc%p = reshape(grid_trim%theta_E,[size(grid_trim%theta_E)])

            ! zeta_F
            grid_1d_loc => grid_1d(id); id = id+1
            grid_1d_loc%var_name = 'zeta_F'
            allocate(grid_1d_loc%tot_i_min(3),grid_1d_loc%tot_i_max(3))
            allocate(grid_1d_loc%loc_i_min(3),grid_1d_loc%loc_i_max(3))
            grid_1d_loc%tot_i_min = [1,1,1]
            grid_1d_loc%tot_i_max = n_tot
            grid_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min]
            grid_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max]
            allocate(grid_1d_loc%p(size(grid_trim%zeta_F)))
            grid_1d_loc%p = reshape(grid_trim%zeta_F,[size(grid_trim%zeta_F)])

            ! zeta_E
            grid_1d_loc => grid_1d(id); id = id+1
            grid_1d_loc%var_name = 'zeta_E'
            allocate(grid_1d_loc%tot_i_min(3),grid_1d_loc%tot_i_max(3))
            allocate(grid_1d_loc%loc_i_min(3),grid_1d_loc%loc_i_max(3))
            grid_1d_loc%tot_i_min = [1,1,1]
            grid_1d_loc%tot_i_max = n_tot
            grid_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min]
            grid_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max]
            allocate(grid_1d_loc%p(size(grid_trim%zeta_E)))
            grid_1d_loc%p = reshape(grid_trim%zeta_E,[size(grid_trim%zeta_E)])
        end if
```

```

        ! write
        ierr = print_hdf5_arrs(grid_1d(1:id-1),pb3d_name,&
            &'grid_'//trim(data_name),rich_lvl=rich_lvl,&
            &ind_print=.not.grid_trim%divided,&
            &remove_previous_arrs=remove_previous_arrs)
        chckerr('')

        ! clean up
        call grid_trim%dealloc()

        ! clean up
        nullify(grid_1d_loc)

        ! user output
        call lvl_ud(-1)
    end function print_output_grid
end module grid_ops
```

D.30 /Users/toonweyens/Code/PB3D/Modules/grid_utilities.f90 File Reference

Interfaces and Types

- interface grid_utilities::coord_f2e

  Converts Flux coordinates $(r,\theta,\zeta)_F$ to Equilibrium coordinates $(r,\theta,\zeta)_E$.
- interface grid_utilities::coord_e2f

  Converts Equilibrium coordinates $(r,\theta,\zeta)_E$. to Flux coordinates $(r,\theta,\zeta)_F$.
- interface grid_utilities::calc_eqd_grid

  Calculate grid of equidistant points, where optionally the last point can be excluded.
- interface grid_utilities::calc_tor_diff

  Calculates the toroidal difference for a magnitude calculated on three toroidal points: two extremities and one in the middle.

Modules

- module grid_utilities

  Numerical utilities related to the grids and different coordinate systems.

Functions/Subroutines

- integer function, public grid_utilities::calc_xyz_grid (grid_eq, grid_xyz, x, y, z, l, r)

  Calculates X, Y and Z on a grid `grid_XYZ`, determined through its E(quilibrium) coordinates.
- integer function, public grid_utilities::extend_grid_f (grid_in, grid_ext, grid_eq, n_theta_plot, n_zeta_plot, lim_theta_plot, lim_zeta_plot)

  Extend a grid angularly.
- integer function, public grid_utilities::copy_grid (grid_a, grid_b, lims_b, i_lim)

  Copy a grid A to a new grid B, that was not yet initialized.
- integer function, public grid_utilities::calc_int_vol (ang_1, ang_2, norm, j, f, f_int)

  Calculates volume integral on a 3D grid.
- integer function, public grid_utilities::trim_grid (grid_in, grid_out, norm_id)

  Trim a grid, removing any overlap between the different regions.
- integer function, public grid_utilities::untrim_grid (grid_in, grid_out, size_ghost)

  Untrims a trimmed grid by introducing an asymmetric ghost regions at the right.

- integer function, public grid_utilities::calc_vec_comp (grid, grid_eq, eq_1, eq_2, v_com, norm_disc_prec, v_mag, base_name, max_transf, v_flux_tor, v_flux_pol, xyz, compare_tor_pos)

  Calculates contra- and covariant components of a vector in multiple coordinate systems.
- integer function, public grid_utilities::calc_n_par_x_rich (n_par_x_rich, only_half_grid)

  Calculates the local number of parallel grid points for this Richardson level, taking into account that it ould be half the actual number.
- integer function, public grid_utilities::nufft (x, f, f_f, plot_name)

  calculates the cosine and sine mode numbers of a function defined on a non-regular grid.
- subroutine, public grid_utilities::find_compr_range (r_f, lim_r, lim_id)

  finds smallest range that comprises a minimum and maximum value.
- integer function, public grid_utilities::calc_arc_angle (grid, eq_1, eq_2, arc, use_e)

  Calculate arclength angle.

Variables

- logical, public grid_utilities::debug_calc_int_vol = .false.

  plot debug information for calc_int_vol()
- logical, public grid_utilities::debug_calc_vec_comp = .false.

  plot debug information for calc_vec_comp()

D.31 grid_utilities.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Numerical utilities related to the grids and different coordinate systems.
!-----------------------------------------------------------------------------!
module grid_utilities
#include <PB3D_macros.h>
#include <wrappers.h>
    use str_utilities
    use output_ops
    use messages
    use num_vars, only: dp, pi, max_str_ln, iu
    use grid_vars, only: grid_type

    implicit none
    private
    public coord_f2e, coord_e2f, calc_xyz_grid, calc_eqd_grid, extend_grid_f, &
        &calc_int_vol, copy_grid, trim_grid, untrim_grid, calc_n_par_x_rich, &
        &calc_vec_comp, nufft, find_compr_range, calc_arc_angle, calc_tor_diff
#if ldebug
    public debug_calc_int_vol, debug_calc_vec_comp
#endif

    ! global variables
#if ldebug
    !> \ldebug
    logical :: debug_calc_int_vol = .false.                                  !< plot debug information for calc_int_vol()
    !> \ldebug
    logical :: debug_calc_vec_comp = .false.                                 !< plot debug information for calc_vec_comp()
#endif

    ! interfaces

    !> \public Converts Flux coordinates
    !!      \f$\left(r,\theta,\zeta\right)_\text{F}\f$
    !!      to Equilibrium coordinates
    !!      \f$\left(r,\theta,\zeta\right)_\text{E}\f$.
    !!
    !! Optionally, two  arrays \c  r_F_array and \c  r_E_array can  be provided,
    !! that define the mapping between the both coordinate system.
    !!
    !! Standard, for E, the poloidal or toroidal normalized flux is used and for
    !! F, the poloidal or toroidal flux in F coordinates, divided by \f$2\pi\f$.
    !!
```

```
    !! \note For VMEC,  it can be slow,  as the zero of  a non-linear expression
    !! must be sought. This is done currently using num_ops.calc_zero_hh().
    !!
    !! \return ierr
    interface coord_f2e
        !> \public
        module procedure coord_f2e_r
        !> \public
        module procedure coord_f2e_rtz
    end interface

    !> \public Converts Equilibrium coordinates
    !!      \f$\left(r,\theta,\zeta\right)_\text{E}\f$.
    !!      to Flux coordinates
    !!      \f$\left(r,\theta,\zeta\right)_\text{F}\f$
    !!
    !! Optionally, two  arrays \c  r_E_array and \c  r_F_array can  be provided,
    !! that define the mapping between the both coordinate system.
    !!
    !! Standard, for E, the poloidal or toroidal normalized flux is used and for
    !! F, the poloidal or toroidal flux in F coordinates, divided by \f$2\pi\f$.
    !!
    !! \return ierr
    interface coord_e2f
        !> \public
        module procedure coord_e2f_r
        !> \public
        module procedure coord_e2f_rtz
    end interface

    !> \public Calculate grid  of equidistant points, where  optionally the last
    !! point can be excluded.
    interface calc_eqd_grid
        !> \public
        module procedure calc_eqd_grid_1d
        !> \public
        module procedure calc_eqd_grid_3d
    end interface

    !> \public Calculates the toroidal difference  for a magnitude calculated on
    !! three toroidal points: two extremities and one in the middle.
    !!
    !! This is done using the formula
    !!  \f[ \frac{b-a}{b+a} , \f]
    !! for the relative  difference between \f$a\f$ and \f$b\f$.  This is useful
    !! when it is used to calculate a  toroidal ripple and these are the extreme
    !! points.
    !!
    !! The procedure also needs the map  between the flux poloidal angle and the
    !! geometrical poloidal angle.
    !!
    !! In a  first step the quantity  is interpolated on an  equidistant grid in
    !! the geometrical poloidal angle.
    !!
    !! The  difference  is then  calculated  for  values of  constant  geometric
    !! poloidal angle.
    !!
    !! Finally, this result is transformed back to the Flux coordinates.
    !!
    !! \note
    !!  -# The theta map should have first and last point equal.
    !!  -# This  routine should  be used  only for  periodic quantities  (so not
    !!  for  some  of the  Flux  quantities  that  are defined  along  generally
    !!  non-rational magnetic field lines).
    !!
    !! \return ierr
    interface calc_tor_diff
        !> \public
        module procedure calc_tor_diff_0d
        !> \public
        module procedure calc_tor_diff_2d
    end interface

contains
    !> \private version withi r, theta and zeta
    integer function coord_f2e_rtz(grid_eq,r_F,theta_F,zeta_F,r_E,&
        &theta_E,zeta_E,r_F_array,r_E_array,ord) result(ierr)
        use num_vars, only: tol_zero, eq_style
        use vmec_utilities, only: fourier2real
        use vmec_vars, only: l_v_c, l_v_s, is_asym_v

        character(*), parameter :: rout_name = 'coord_F2E_rtz'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                              !< equilibrium grid (for normal local limits and possibly loc_r)
        real(dp), intent(in) :: r_f(:)                                      !< (local) \f$r_\text{F}\f$
```

```
        real(dp), intent(in) :: theta_f(:,:,:)                          !< \f$\theta_\text{F}\f$
        real(dp), intent(in) :: zeta_f(:,:,:)                           !< \f$\zeta_\text{F}\f$
        real(dp), intent(inout) :: r_e(:)                               !< (local) \f$r_\text{E}\f$
        real(dp), intent(inout) :: theta_e(:,:,:)                       !< \f$\theta_\text{E}\f$
        real(dp), intent(inout) :: zeta_e(:,:,:)                        !< \f$\zeta_\text{E}\f$
        real(dp), intent(in), optional, target :: r_f_array(:)          !< optional array that defines mapping between two coord. systems
        real(dp), intent(in), optional, target :: r_e_array(:)          !< optional array that defines mapping between two coord. systems
        integer, intent(in), optional :: ord                            !< order (only used for eq_style 1)

        ! local variables (also used in child functions)
        character(len=max_str_ln) :: err_msg                            ! error message
        integer :: dims(3)                                              ! dimensions of the grid
        real(dp), allocatable :: l_v_c_loc(:,:)                         ! local version of L_V_c
        real(dp), allocatable :: l_v_s_loc(:,:)                         ! local version of L_V_s
        integer :: ord_loc                                              ! local ord

        ! initialize ierr
        ierr = 0

        ! set up array sizes
        dims = shape(theta_e)

        ! set up local ord
        ord_loc = 1
        if (present(ord)) ord_loc = ord

        ! tests
        if (dims(3).ne.size(r_f) .or. dims(3).ne.size(r_e)) then
            ierr = 1
            err_msg = 'r_F and r_E need to have the correct dimensions'
            chckerr(err_msg)
        end if
        if (present(r_f_array).neqv.present(r_e_array)) then
            ierr = 1
            err_msg = 'both r_F_array and r_E_array have to be provided'
            chckerr(err_msg)
        end if

        ! convert normal position
        ierr = coord_f2e_r(grid_eq,r_f,r_e,r_f_array,r_e_array)
        chckerr('')

        ! choose which equilibrium style is being used:
        !   1:  VMEC
        !   2:  HELENA
        select case (eq_style)
            case (1)                                                    ! VMEC
                ierr = coord_f2e_vmec()
                chckerr('')
            case (2)                                                    ! HELENA
                ! trivial HELENA uses flux coordinates
                theta_e = theta_f
                zeta_e = zeta_f
        end select
    contains
        ! VMEC version
        !> \private
        integer function coord_f2e_vmec() result(ierr)
            use num_vars, only: norm_disc_prec_eq, rank
            use num_ops, only: calc_zero_hh
            use vmec_vars, only: mnmax_v
            use num_utilities, only: spline

            character(*), parameter :: rout_name = 'coord_F2E_VMEC'

            ! local variables
            integer :: id                                               ! counter
            real(dp), pointer :: loc_r_f(:) => null()                   ! loc_r in F coords.
            real(dp), allocatable :: theta_e_guess_3d(:,:,:)            ! guess for theta_E
            real(dp), allocatable :: lam(:,:,:,:)                       ! lambda and derivative

            ! initialize ierr
            ierr = 0

            ! set up loc_r_F
            if (present(r_f_array)) then
                loc_r_f => r_f_array
            else
                loc_r_f => grid_eq%loc_r_F
            end if

            ! the toroidal angle is trivial
            zeta_e = - zeta_f                                           ! conversion VMEC LH -> RH coord. system

            ! allocate local copies of L_V_c and L_V_s
```

```
            allocate(l_v_c_loc(mnmax_v,dims(3)))
            allocate(l_v_s_loc(mnmax_v,dims(3)))

            ! interpolate L_V_c and L_V_s at requested normal points r_F
            do id = 1,mnmax_v
                ierr = spline(loc_r_f,l_v_c(id,grid_eq%i_min:grid_eq%i_max,0),&
                    &r_f,l_v_c_loc(id,:),ord=norm_disc_prec_eq)
                chckerr('')
                ierr = spline(loc_r_f,l_v_s(id,grid_eq%i_min:grid_eq%i_max,0),&
                    &r_f,l_v_s_loc(id,:),ord=norm_disc_prec_eq)
                chckerr('')
            end do

            ! set up guess
            allocate(theta_e_guess_3d(dims(1),dims(2),dims(3)))
            allocate(lam(dims(1),dims(2),dims(3),0:1))

            ! set up guess: try theta_F - lambda(theta_F)/(1+Dlambda(theta_F)),
            ! which is what you get when you approximate lambda(theta_V) as
            ! lambda(theta_F) + Dlambda(theta_F) (theta_V-theta_F)
            ! (lambda(theta_V) would be the exact solution)
            do id = 0,1
                ierr = fourier2real(l_v_c_loc,l_v_s_loc,theta_f,zeta_e,&
                    &lam(:,:,:,id),sym=[is_asym_v,.true.],deriv=[id,0])
                chckerr('')
            end do
            theta_e_guess_3d = theta_f - lam(:,:,:,0)/(1+lam(:,:,:,1))

            ! the poloidal angle has to be found as the zero of
            !   f = theta_F - theta_E - lambda
            err_msg = calc_zero_hh(dims,theta_e,fun_pol,ord_loc,&
                &theta_e_guess_3d,output=.true.)
            if (err_msg.ne.'') then
                ierr = 1
                chckerr(err_msg)
            end if

            ! do a check whether the result is indeed zero
            if (maxval(abs(fun_pol(dims,theta_e,0))).gt.tol_zero*100) then
                call plot_hdf5('f','ERROR_coord_F2E_VMEC_'//trim(i2str(rank)),&
                    &fun_pol(dims,theta_e,0))
                err_msg = 'In coord_F2E_VMEC, calculating whether f=0 as a &
                    &check, yields a deviation from 0 of '//trim(r2strt(&
                    &maxval(abs(fun_pol(dims,theta_e,0))))//'. See output.'
                ierr = 1
                chckerr(err_msg)
            end if

            ! clean up
            nullify(loc_r_f)
        end function coord_f2e_vmec

        ! function  that  returns  f  =  theta_F  -  theta_V  -  lambda  or  its
        ! derivatives in the poloidal direction.  It uses zeta_E (= zeta_V) from
        ! the parent function.
        !> \private
        function fun_pol(dims,theta_E_in,dpol)
            character(*), parameter :: rout_name = 'fun_pol'

            ! input / output
            integer, intent(in) :: dims(3)
            real(dp), intent(in) :: theta_e_in(dims(1),dims(2),dims(3))
            integer, intent(in) :: dpol
            real(dp) :: fun_pol(dims(1),dims(2),dims(3))

            ! local variables
            real(dp) :: lam(dims(1),dims(2),dims(3))

            ! initialize fun_pol
            fun_pol = 0.0_dp

            ! calculate lambda
            ierr = fourier2real(l_v_c_loc,l_v_s_loc,theta_e_in,zeta_e,lam,&
                &sym=[is_asym_v,.true.],deriv=[dpol,0])
            chckerr('')

            ! calculate the output function
            if (dpol.eq.0) then
                fun_pol = theta_f - theta_e_in - lam
            else if (dpol.eq.1) then
                fun_pol = -1 - lam
            else if (dpol.gt.1) then
                fun_pol = - lam
            end if
        end function fun_pol
    end function coord_f2e_rtz
    !> \private version with only r
```

```
    integer function coord_f2e_r(grid_eq,r_F,r_E,r_F_array,r_E_array) &
        &result(ierr)                                                   ! version with only r
        use num_vars, only: norm_disc_prec_eq
        use num_utilities, only: spline

        character(*), parameter :: rout_name = 'coord_F2E_r'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                          !< equilibrium grid (for possibly loc_r)
        real(dp), intent(in) :: r_f(:)                                  !< (local) \f$r_\text{F}\f$
        real(dp), intent(inout) :: r_e(:)                               !< (local) \f$r_\text{E}\f$
        real(dp), intent(in), optional, target :: r_f_array(:)          !< optional array that defines mapping between two coord. systems
        real(dp), intent(in), optional, target :: r_e_array(:)          !< optional array that defines mapping between two coord. systems

        ! local variables
        character(len=max_str_ln) :: err_msg                            ! error message
        integer :: n_r                                                  ! dimension of the grid
        real(dp), pointer :: loc_r_e(:) => null()                       ! loc_r in E coords.
        real(dp), pointer :: loc_r_f(:) => null()                       ! loc_r in F coords.

        ! initialize ierr
        ierr = 0

        ! set up array sizes
        n_r = size(r_f)

        ! tests
        if (n_r.ne.size(r_e)) then
            ierr = 1
            err_msg = 'r_F and r_E need to have the correct dimensions'
            chckerr(err_msg)
        end if
        if (present(r_f_array).neqv.present(r_e_array)) then
            ierr = 1
            err_msg = 'both r_F_array and r_E_array have to be provided'
            chckerr(err_msg)
        end if

        ! set up loc_r_E and loc_r_F
        if (present(r_f_array)) then
            loc_r_e => r_e_array
            loc_r_f => r_f_array
        else
            loc_r_e => grid_eq%loc_r_E
            loc_r_f => grid_eq%loc_r_F
        end if

        ! convert normal position
        ierr = spline(loc_r_f,loc_r_e,r_f,r_e,ord=norm_disc_prec_eq)
        chckerr('')

        ! clean up
        nullify(loc_r_e,loc_r_f)
    end function coord_f2e_r

    !> version with r, theta and zeta
    integer function coord_e2f_rtz(grid_eq,r_E,theta_E,zeta_E,r_F,&
        &theta_F,zeta_F,r_E_array,r_F_array) result(ierr)               ! version with r, theta and zeta
        use num_vars, only: eq_style

        character(*), parameter :: rout_name = 'coord_E2F_rtz'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                          !< equilibrium grid (for normal local limits)
        real(dp), intent(in) :: r_e(:)                                  !< (local) \f$r_\text{E}\f$
        real(dp), intent(in) :: theta_e(:,:,:)                          !< \f$\theta_\text{E}\f$
        real(dp), intent(in) :: zeta_e(:,:,:)                           !< \f$\zeta_\text{E}\f$
        real(dp), intent(inout) :: r_f(:)                               !< (local) \f$r_\text{F}\f$
        real(dp), intent(inout) :: theta_f(:,:,:)                       !< \f$\theta_\text{F}\f$
        real(dp), intent(inout) :: zeta_f(:,:,:)                        !< \f$\zeta_\text{F}\f$
        real(dp), intent(in), optional, target :: r_e_array(:)          !< optional array that defines mapping between two coord. systems
        real(dp), intent(in), optional, target :: r_f_array(:)          !< optional array that defines mapping between two coord. systems

        ! local variables (also used in child functions)
        character(len=max_str_ln) :: err_msg                            ! error message
        integer :: dims(3)                                              ! dimensions of the grid

        ! initialize ierr
        ierr = 0

        ! set up array sizes
        dims = shape(theta_f)
```

```

        ! tests
        if (dims(3).ne.size(r_f) .or. dims(3).ne.size(r_e)) then
            ierr = 1
            err_msg = 'r_F and r_E need to have the correct dimensions'
            chckerr(err_msg)
        end if
        if (present(r_f_array).neqv.present(r_e_array)) then
            ierr = 1
            err_msg = 'both r_F_array and r_E_array have to be provided'
            chckerr(err_msg)
        end if

        ! convert normal position
        ierr = coord_e2f_r(grid_eq,r_e,r_f,r_e_array,r_f_array)
        chckerr('')

        ! choose which equilibrium style is being used:
        !   1:  VMEC
        !   2:  HELENA
        select case (eq_style)
            case (1)                                                        ! VMEC
                ierr = coord_e2f_vmec()
                chckerr('')
            case (2)                                                        ! HELENA
                ! trivial HELENA uses flux coordinates
                theta_f = theta_e
                zeta_f = zeta_e
        end select
    contains
        ! VMEC version
        !> \private
        integer function coord_e2f_vmec() result(ierr)
            use num_vars, only: norm_disc_prec_eq
            use vmec_utilities, only: fourier2real
            use vmec_vars, only: mnmax_v, l_v_c, l_v_s, is_asym_v
            use num_utilities, only: spline

            character(*), parameter :: rout_name = 'coord_E2F_VMEC'

            ! local variables
            integer :: id                                                   ! counter
            real(dp), allocatable :: l_v_c_loc(:,:), l_v_s_loc(:,:)         ! local version of L_V_c and L_V_s
            real(dp), allocatable :: lam(:,:,:)                             ! lambda
            real(dp), pointer :: loc_r_e(:) => null()                       ! loc_r in E coords.

            ! initialize ierr
            ierr = 0

            ! set up loc_r_E
            if (present(r_e_array)) then
                loc_r_e => r_e_array
            else
                loc_r_e => grid_eq%loc_r_E
            end if

            ! the toroidal angle is trivial
            zeta_f = - zeta_e                                               ! conversion VMEC LH -> RH coord. system

            ! allocate local copies of L_V_c and L_V_s and lambda
            allocate(l_v_c_loc(mnmax_v,dims(3)))
            allocate(l_v_s_loc(mnmax_v,dims(3)))
            allocate(lam(dims(1),dims(2),dims(3)))

            ! interpolate L_V_c and L_V_s at requested normal points r_E
            do id = 1,mnmax_v
                ierr = spline(loc_r_e,l_v_c(id,grid_eq%i_min:grid_eq%i_max,0),&
                    &r_e,l_v_c_loc(id,:),ord=norm_disc_prec_eq)
                chckerr('')
                ierr = spline(loc_r_e,l_v_s(id,grid_eq%i_min:grid_eq%i_max,0),&
                    &r_e,l_v_s_loc(id,:),ord=norm_disc_prec_eq)
                chckerr('')
            end do

            ! calculate lambda
            ierr = fourier2real(l_v_c_loc,l_v_s_loc,theta_e,zeta_e,lam,&
                &sym=[is_asym_v,.true.])
            chckerr('')

            ! the poloidal angle has to be found as
            !   theta_F = theta_E + lambda
            theta_f = theta_e + lam

            ! clean up
            nullify(loc_r_e)
        end function coord_e2f_vmec
    end function coord_e2f_rtz
```

```
    !> version with only r
    integer function coord_e2f_r(grid_eq,r_E,r_F,r_E_array,r_F_array) &
        &result(ierr)                                                   ! version with only r
        use num_vars, only: norm_disc_prec_eq
        use num_utilities, only: spline

        character(*), parameter :: rout_name = 'coord_E2F_r'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                          !< equilibrium grid (for normal local limits)
        real(dp), intent(in) :: r_e(:)                                  !< (local) \f$r_\text{E}\f$
        real(dp), intent(inout) :: r_f(:)                               !< (local) \f$r_\text{F}\f$
        real(dp), intent(in), optional, target :: r_e_array(:)          !< optional array that defines mapping between two coord. systems
        real(dp), intent(in), optional, target :: r_f_array(:)          !< optional array that defines mapping between two coord. systems

        ! local variables
        character(len=max_str_ln) :: err_msg                            ! error message
        integer :: n_r                                                  ! dimension of the grid
        real(dp), pointer :: loc_r_e(:) => null()                       ! loc_r in E coords.
        real(dp), pointer :: loc_r_f(:) => null()                       ! loc_r in F coords.

        ! initialize ierr
        ierr = 0

        ! set up array sizes
        n_r = size(r_e)

        ! tests
        if (n_r.ne.size(r_f)) then
            ierr = 1
            err_msg = 'r_E and r_F need to have the correct dimensions'
            chckerr(err_msg)
        end if
        if (present(r_e_array).neqv.present(r_f_array)) then
            ierr = 1
            err_msg = 'both r_E_array and r_F_array have to be provided'
            chckerr(err_msg)
        end if

        ! set up loc_r_E and loc_r_F
        if (present(r_f_array)) then
            loc_r_e => r_e_array
            loc_r_f => r_f_array
        else
            loc_r_e => grid_eq%loc_r_E
            loc_r_f => grid_eq%loc_r_F
        end if

        ! convert normal position
        ierr = spline(loc_r_e,loc_r_f,r_e,r_f,ord=norm_disc_prec_eq)
        chckerr('')

        ! clean up
        nullify(loc_r_e,loc_r_f)
    end function coord_e2f_r

    !> \private 3-D version
    integer function calc_eqd_grid_3d(var,min_grid,max_grid,grid_dim,&
        &excl_last) result(ierr)
        character(*), parameter :: rout_name = 'calc_eqd_grid_3D'

        ! input and output
        real(dp), intent(inout) :: var(:,:,:)                           !< output
        real(dp), intent(in) :: min_grid                                !< min. of angles [\f$\pi\f$]
        real(dp), intent(in) :: max_grid                                !< max. of angles [\f$\pi\f$]
        integer, intent(in) :: grid_dim                                 !< in which dimension to create the grid
        logical, intent(in), optional :: excl_last                      !< .true. if last point excluded

        ! local variables
        integer :: id                                                   ! counter
        character(len=max_str_ln) :: err_msg                            ! error message
        logical :: excl_last_loc                                        ! local copy of excl_last
        integer :: grid_size                                            ! nr. of points
        integer :: grid_size_mod                                        ! grid_size or grid_size + 1

        ! initialize ierr
        ierr = 0

        ! tests
        if (grid_dim.lt.1 .or. grid_dim.gt.3) then
            ierr = 1
            err_msg = 'grid_dim has to point to a dimension going from 1 to 3'
            chckerr(err_msg)
        end if
```

```

        ! set up grid_size
        grid_size = size(var,grid_dim)

        ! test some values
        if (grid_size.lt.1) then
            err_msg = 'The angular array has to have a length of at &
                &least 1'
            ierr = 1
            chckerr(err_msg)
        end if

        ! set up local excl_last
        excl_last_loc = .false.
        if (present(excl_last)) excl_last_loc = excl_last

        ! add 1 to modified grid size if last point is to be excluded
        if (excl_last_loc) then
            grid_size_mod = grid_size + 1
        else
            grid_size_mod = grid_size
        end if

        ! initialize output vector
        var = 0.0_dp

        ! calculate grid points
        if (grid_size.eq.1) then
            var = min_grid                                                  ! = max_grid
        else
            if (grid_dim.eq.1) then
                do id = 1,grid_size
                    var(id,:,:) = min_grid + &
                        &(max_grid-min_grid)*(id-1)/(grid_size_mod-1)
                end do
            else if (grid_dim.eq.2) then
                do id = 1,grid_size
                    var(:,id,:) = min_grid + &
                        &(max_grid-min_grid)*(id-1)/(grid_size_mod-1)
                end do
            else
                do id = 1,grid_size
                    var(:,:,id) = min_grid + &
                        &(max_grid-min_grid)*(id-1)/(grid_size_mod-1)
                end do
            end if
        end if
    end function calc_eqd_grid_3d
    !> \private 1-D version
    integer function calc_eqd_grid_1d(var,min_grid,max_grid,excl_last) &
        &result(ierr)
        character(*), parameter :: rout_name = 'calc_eqd_grid_1D'

        ! input and output
        real(dp), intent(inout) :: var(:)                                   !< output
        real(dp), intent(in) :: min_grid                                    !< min. of angles [\f$\pi\f$]
        real(dp), intent(in) :: max_grid                                    !< max. of angles [\f$\pi\f$]
        logical, intent(in), optional :: excl_last                          !< .true. if last point excluded

        ! local variables
        real(dp), allocatable :: var_3d(:,:,:)                              ! 3D version of var

        ! initialize ierr
        ierr = 0

        ! set up var_3D
        allocate(var_3d(size(var),1,1))

        ! call 3D version
        ierr = calc_eqd_grid_3d(var_3d,min_grid,max_grid,1,excl_last)
        chckerr('')

        ! update var
        var = var_3d(:,1,1)
    end function calc_eqd_grid_1d

    !> \private 2-D version
    integer function calc_tor_diff_2d(v_com,theta,norm_disc_prec,absolute,r) &
        &result(ierr)
        use num_vars, only: min_theta_plot, max_theta_plot, tol_zero
        use num_utilities, only: spline, order_per_fun

        character(*), parameter :: rout_name = 'calc_tor_diff_2D'

        ! input / output
        real(dp), intent(inout) :: v_com(:,:,:,:,:)                         !< covariant and contravariant components
<tt>(dim1,dim2,dim3,3,2)</tt>
```

```
        real(dp), intent(in) :: theta(:,:,:)                                !< geometric poloidal angle
        integer, intent(in) :: norm_disc_prec                               !< precision for normal derivatives
        logical, intent(in), optional :: absolute                           !< calculate absolute, not relative, difference
        real(dp), intent(in), optional :: r(:)                              !< normal positions for theta

        ! local variables
        integer :: id, jd, kd, ld                                           ! counters
        integer :: n_pol                                                    ! number of poloidal points to be used (1 less than total)
        integer :: istat                                                    ! status
        real(dp), allocatable :: theta_eqd(:)                               ! equidistant grid
        real(dp), allocatable :: v_com_interp(:,:)                          ! interpolated local v_com for outer points
        real(dp), allocatable :: theta_ord(:)                               ! ordered theta
        real(dp), allocatable :: v_com_ord(:)                               ! ordered v_com
        logical :: absolute_loc                                             ! local absolute

        ! initialize ierr
        ierr = 0

        ! set up variables
        n_pol = size(theta,1)-1
        allocate(theta_eqd(n_pol))
        allocate(v_com_interp(n_pol,3))
        absolute_loc = .false.
        if (present(absolute)) absolute_loc = absolute

        norm: do kd = 1,size(v_com,3)
            ! set up equidistant grid
            ierr = calc_eqd_grid(theta_eqd,min_theta_plot*pi,max_theta_plot*pi,&
                &excl_last=.true.)
            chckerr('')
            do id = 1,size(v_com,4)
                do ld = 1,size(v_com,5)
                    ! interpolate the geometric poloidal angle for outer points
                    do jd = 1,3,2
                        ! order
                        ierr = order_per_fun(theta(1:n_pol,jd,kd),&
                            &v_com(1:n_pol,jd,kd,id,ld),theta_ord,v_com_ord,&
                            &norm_disc_prec)
                        chckerr('')

                        istat = spline(theta_ord,v_com_ord,theta_eqd,&
                            &v_com_interp(:,jd),ord=min(norm_disc_prec,3),&
                            &deriv=0)
                        deallocate(theta_ord,v_com_ord)

                        if (istat.ne.0) then
                            call display_interp_warning(r)
                            v_com(:,2,kd,:,:) = 0._dp
                            cycle norm
                        end if
                    end do

                    ! calculate difference and save in middle point
                    v_com_interp(:,2) = &
                        &(v_com_interp(:,3)-v_com_interp(:,1))
                    if (.not.absolute_loc) &                                ! make it relative
                        &v_com_interp(:,2) = v_com_interp(:,2)/&
                        &sign(max(&
                        &tol_zero,abs(v_com_interp(:,3)+v_com_interp(:,1))),&
                        &v_com_interp(:,3)+v_com_interp(:,1))

                    ! order
                    ierr = order_per_fun(theta_eqd,v_com_interp(:,2),&
                        &theta_ord,v_com_ord,norm_disc_prec)
                    chckerr('')

                    ! interpolate back
                    istat = spline(theta_ord,v_com_ord,theta(:,2,kd),&
                        &v_com(:,2,kd,id,ld),min(norm_disc_prec,3),&
                        &deriv=0)
                    deallocate(theta_ord,v_com_ord)

                    if (istat.ne.0) then
                        call display_interp_warning(r)
                        v_com(:,2,kd,:,:) = 0._dp
                        cycle norm
                    end if
                end do
            end do
        end do norm
    contains
        ! Displays warning if no interpolation possible.
        ! Uses variable kd from parent procedure.
        !> \private
        subroutine display_interp_warning(r)
```

```
            ! input / output
            real(dp), intent(in), optional :: r(:)                              ! normal positions for theta

            if (present(r)) then
                call writo('Cannot interpolate geometrical &
                    &theta at normal position '//&
                    &trim(r2str(r(kd))),warning=.true.)
            else
                call writo('Cannot interpolate geometrical &
                    &theta',warning=.true.)
            end if
            call lvl_ud(1)
            call writo('Are you sure the origin is chosen &
                &correctly?')
            call writo('Skipping this normal position')
            call lvl_ud(-1)
        end subroutine display_interp_warning
    end function calc_tor_diff_2d
    !> \private 0-D version
    integer function calc_tor_diff_0d(v_mag,theta,norm_disc_prec,absolute,r) &
        &result(ierr)
        character(*), parameter :: rout_name = 'calc_tor_diff_0D'

        ! input / output
        real(dp), intent(inout) :: v_mag(:,:,:)                                 !< magnitude <tt>(dim1,dim2,dim3)</tt>
        real(dp), intent(in) :: theta(:,:,:)                                    !< geometric poloidal angle
        integer, intent(in) :: norm_disc_prec                                   !< precision for normal derivatives
        logical, intent(in), optional :: absolute                               !< calculate absolute, not relative, difference
        real(dp), intent(in), optional :: r(:)                                  !< normal positions for theta

        ! local variable
        real(dp), allocatable :: v_com_loc(:,:,:,:,:)                           ! local copy of v_mag

        ! initialize ierr
        ierr = 0

        ! set up local copy of v_mag
        allocate(v_com_loc(size(v_mag,1),size(v_mag,2),size(v_mag,3),1,1))
        v_com_loc(:,:,:,1,1) = v_mag

        ! call 2-D version
        ierr = calc_tor_diff_2d(v_com_loc,theta,norm_disc_prec,&
            &absolute=absolute,r=r)
        chckerr('')

        ! copy
        v_mag = v_com_loc(:,:,:,1,1)
    end function calc_tor_diff_0d

    !> Calculates \f$X\f$, \f$Y\f$ and \f$Z\f$ on a grid \c grid_XYZ, determined
    !! through its E(quilibrium) coordinates.
    !!
    !! Furthermore, a  grid \c grid_eq  must be provided,  which is the  grid in
    !! which the  variables concerning \f$R\f$  and \f$Z\f$ are  tabulated, i.e.
    !! the full equilibrium grid in E(quilibrium) coordinates.
    !!
    !! Of this grid, however, only \c r_E  is used, and the rest ignored. It can
    !! therefore be provided without the angular part, i.e. by reconstructing it
    !! with a subset.
    !!
    !! If VMEC is the equilibrium model, this routine also optionally calculates
    !! \f$\lambda\f$ on the grid, as this  is also needed some times. for HELENA
    !! this variable is not used.
    !!
    !! Optionally, \f$R\f$ and \f$\lambda\f$ (only for VMEC) can be returned.
    !!
    !! \note
    !!  -# For VMEC, the trigonometric factors of \c grid_XYZ must be calculated
    !!  beforehand.
    !!  -# The normalization factor \c R_0  for length is taken into account and
    !!  the output is transformed back to unnormalized values.
    !!
    !! \return ierr
    integer function calc_xyz_grid(grid_eq,grid_XYZ,X,Y,Z,L,R) result(ierr)
        use num_vars, only: eq_style, use_normalization
        use num_utilities, only: round_with_tol
        use eq_vars, only: r_0

        character(*), parameter :: rout_name = 'calc_XYZ_grid'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(grid_type), intent(in) :: grid_xyz                                 !< grid for which to calculate \f$X\f$, \f$Y\f$, \f$Z\f$ and optionally \f$L\f$
        real(dp), intent(inout) :: x(:,:,:)                                     !< \f$X\f$ of grid
        real(dp), intent(inout) :: y(:,:,:)                                     !< \f$Y\f$ of grid
```

```
        real(dp), intent(inout) :: z(:,:,:)                                     !< \f$Z\f$ of grid
        real(dp), intent(inout), optional :: l(:,:,:)                           !< \f$\lambda\f$ of grid
        real(dp), intent(inout), optional :: r(:,:,:)                           !< \f$R\f$ of grid

        ! local variables
        character(len=max_str_ln) :: err_msg                                    ! error message

        ! initialize ierr
        ierr = 0

        ! test
        if (size(x,1).ne.grid_xyz%n(1) .or. size(x,2).ne.grid_xyz%n(2) .or. &
            &size(x,3).ne.grid_xyz%loc_n_r) then
            ierr = 1
            err_msg =  'X needs to have the correct dimensions'
            chckerr(err_msg)
        end if
        if (size(y,1).ne.grid_xyz%n(1) .or. size(y,2).ne.grid_xyz%n(2) .or. &
            &size(y,3).ne.grid_xyz%loc_n_r) then
            ierr = 1
            err_msg =  'Y needs to have the correct dimensions'
            chckerr(err_msg)
        end if
        if (size(z,1).ne.grid_xyz%n(1) .or. size(z,2).ne.grid_xyz%n(2) .or. &
            &size(z,3).ne.grid_xyz%loc_n_r) then
            ierr = 1
            err_msg =  'Z needs to have the correct dimensions'
            chckerr(err_msg)
        end if
        if (present(l)) then
            if (size(l,1).ne.grid_xyz%n(1) .or. size(l,2).ne.grid_xyz%n(2) &
                &.or. size(l,3).ne.grid_xyz%loc_n_r) then
                ierr = 1
                err_msg =  'L needs to have the correct dimensions'
                chckerr(err_msg)
            end if
        end if

        ! choose which equilibrium style is being used:
        !   1:  VMEC
        !   2:  HELENA
        select case (eq_style)
            case (1)                                                            ! VMEC
                ierr = calc_xyz_grid_vmec(grid_eq,grid_xyz,x,y,z,l=l,r=r)
                chckerr('')
            case (2)                                                            ! HELENA
                ierr = calc_xyz_grid_hel(grid_eq,grid_xyz,x,y,z,r=r)
                chckerr('')
        end select

        ! if normalized, return to physical value
        if (use_normalization) then
            x = x*r_0
            y = y*r_0
            z = z*r_0
            if (present(r)) r = r*r_0
        end if
    contains
        ! VMEC version
        !> \private
        integer function calc_xyz_grid_vmec(grid_eq,grid_XYZ,X,Y,Z,L,R) &
            &result(ierr)
            use num_vars, only: norm_disc_prec_eq
            use vmec_utilities, only: fourier2real
            use vmec_vars, only: r_v_c, r_v_s, z_v_c, z_v_s, l_v_c, l_v_s, &
                &mnmax_v, is_asym_v
            use num_utilities, only: spline

            character(*), parameter :: rout_name = 'calc_XYZ_grid_VMEC'

            ! input / output
            type(grid_type), intent(in) :: grid_eq                              ! equilibrium grid
            type(grid_type), intent(in) :: grid_xyz                             ! grid for which to calculate X, Y, Z and
    optionally L
            real(dp), intent(inout) :: x(:,:,:), y(:,:,:), z(:,:,:)             ! X, Y and Z of grid
            real(dp), intent(inout), optional :: l(:,:,:)                       ! lambda of grid
            real(dp), intent(inout), optional :: r(:,:,:)                       ! R of grid

            ! local variables
            integer :: id                                                       ! counter
            real(dp), allocatable :: r_v_c_int(:,:), r_v_s_int(:,:)             ! interpolated version of R_V_c and R_V_s
            real(dp), allocatable :: z_v_c_int(:,:), z_v_s_int(:,:)             ! interpolated version of Z_V_c and Z_V_s
            real(dp), allocatable :: l_v_c_int(:,:), l_v_s_int(:,:)             ! interpolated version of L_V_c and L_V_s
            real(dp), allocatable :: r_loc(:,:,:)                               ! R in Cylindrical coordinates
            character(len=max_str_ln) :: err_msg                                ! error message

            ! initialize ierr
```

```
            ierr = 0

            ! test
            if (.not.allocated(grid_xyz%trigon_factors)) then
                ierr = 1
                err_msg = 'trigon factors of grid_XYZ need to be allocated'
                chckerr(err_msg)
            end if

            ! set up interpolated R_V_c_int, ..
            allocate(r_v_c_int(mnmax_v,grid_xyz%loc_n_r))
            allocate(r_v_s_int(mnmax_v,grid_xyz%loc_n_r))
            allocate(z_v_c_int(mnmax_v,grid_xyz%loc_n_r))
            allocate(z_v_s_int(mnmax_v,grid_xyz%loc_n_r))
            if (present(l)) then
                allocate(l_v_c_int(mnmax_v,grid_xyz%loc_n_r))
                allocate(l_v_s_int(mnmax_v,grid_xyz%loc_n_r))
            end if

            ! interpolate VMEC tables
            do id = 1,mnmax_v
                ierr = spline(grid_eq%r_E,r_v_c(id,:,0),grid_xyz%loc_r_E,&
                    &r_v_c_int(id,:),ord=norm_disc_prec_eq)
                chckerr('')
                ierr = spline(grid_eq%r_E,r_v_s(id,:,0),grid_xyz%loc_r_E,&
                    &r_v_s_int(id,:),ord=norm_disc_prec_eq)
                chckerr('')
                ierr = spline(grid_eq%r_E,z_v_c(id,:,0),grid_xyz%loc_r_E,&
                    &z_v_c_int(id,:),ord=norm_disc_prec_eq)
                chckerr('')
                ierr = spline(grid_eq%r_E,z_v_s(id,:,0),grid_xyz%loc_r_E,&
                    &z_v_s_int(id,:),ord=norm_disc_prec_eq)
                chckerr('')
                if (present(l)) then
                    ierr = spline(grid_eq%r_E,l_v_c(id,:,0),grid_xyz%loc_r_E,&
                        &l_v_c_int(id,:),ord=norm_disc_prec_eq)
                    chckerr('')
                    ierr = spline(grid_eq%r_E,l_v_s(id,:,0),grid_xyz%loc_r_E,&
                        &l_v_s_int(id,:),ord=norm_disc_prec_eq)
                    chckerr('')
                end if
            end do

            ! allocate local R
            allocate(r_loc(grid_xyz%n(1),grid_xyz%n(2),grid_xyz%loc_n_r))

            ! inverse fourier transform with trigonometric factors
            ierr = fourier2real(r_v_c_int,r_v_s_int,grid_xyz%trigon_factors,&
                &r_loc,sym=[.true.,is_asym_v])
            chckerr('')
            ierr = fourier2real(z_v_c_int,z_v_s_int,grid_xyz%trigon_factors,z,&
                &sym=[is_asym_v,.true.])
            chckerr('')
            if (present(l)) then
                ierr = fourier2real(l_v_c_int,l_v_s_int,&
                    &grid_xyz%trigon_factors,l,sym=[is_asym_v,.true.])
                chckerr('')
            end if
            if (present(r)) r = r_loc

            ! transform cylindrical to cartesian
            ! (the geometrical zeta is equal to the VMEC zeta)
            x = r_loc*cos(grid_xyz%zeta_E)
            y = r_loc*sin(grid_xyz%zeta_E)

            ! deallocate
            deallocate(r_v_c_int,r_v_s_int,z_v_c_int,z_v_s_int)
            if (present(l)) deallocate(l_v_c_int,l_v_s_int)
            deallocate(r_loc)
        end function calc_xyz_grid_vmec

        ! HELENA version
        !> \private
        integer function calc_xyz_grid_hel(grid_eq,grid_XYZ,X,Y,Z,R) &
            &result(ierr)
            use helena_vars, only: r_h, z_h, chi_h, ias, nchi
            use num_vars, only: norm_disc_prec_eq
            use num_utilities, only: spline

            character(*), parameter :: rout_name = 'calc_XYZ_grid_HEL'

            ! input / output
            type(grid_type), intent(in) :: grid_eq                          ! equilibrium grid
            type(grid_type), intent(in) :: grid_xyz                         ! grid for which to calculate X, Y, Z and optionally L
            real(dp), intent(inout) :: x(:,:,:), y(:,:,:), z(:,:,:)         ! X, Y and Z of grid
            real(dp), intent(inout), optional :: r(:,:,:)                   ! R of grid
```

```
            ! local variables
            integer :: id, jd, kd                                          ! counters
            integer :: pmone                                               ! plus or minus one
            integer :: bcs(2,3)                                            ! boundary conditions (theta(even), theta(odd), r)
            real(dp) :: bcs_val(2,3)                                       ! values for boundary conditions
            real(dp), allocatable :: r_h_int(:,:), z_h_int(:,:)            ! R and Z at interpolated normal value
            real(dp), allocatable :: r_loc(:,:,:)                          ! R in Cylindrical coordinates
            real(dp) :: theta_loc                                          ! local copy of theta_E

            ! initialize ierr
            ierr = 0

            ! set up boundary conditions
            if (ias.eq.0) then                                             ! top-bottom symmetric
                bcs(:,1) = [1,1]                                           ! theta(even): zero first derivative
                bcs(:,2) = [2,2]                                           ! theta(odd): zero first derivative
            else
                bcs(:,1) = [-1,-1]                                         ! theta(even): periodic
                bcs(:,2) = [-1,-1]                                         ! theta(odd): periodic
            end if
            bcs(:,3) = [0,0]                                               ! r: not-a-knot
            bcs_val = 0._dp

            ! allocate local R
            allocate(r_loc(grid_xyz%n(1),grid_xyz%n(2),grid_xyz%loc_n_r))

            ! set up interpolated R and Z
            allocate(r_h_int(nchi,grid_xyz%loc_n_r))
            allocate(z_h_int(nchi,grid_xyz%loc_n_r))

            ! interpolate HELENA output  R_H and Z_H for  every requested normal
            ! point
            do id = 1,nchi
                ierr = spline(grid_eq%r_E,r_h(id,:),grid_xyz%loc_r_E,&
                    &r_h_int(id,:),ord=norm_disc_prec_eq,bcs=bcs(:,3),&
                    &bcs_val=bcs_val(:,3))
                chckerr('')
                ierr = spline(grid_eq%r_E,z_h(id,:),grid_xyz%loc_r_E,&
                    &z_h_int(id,:),ord=norm_disc_prec_eq,bcs=bcs(:,3),&
                    &bcs_val=bcs_val(:,3))
                chckerr('')
            end do

            ! Note:  R_H and  Z_H  are not  adapted to  the  parallel grid,  but
            ! tabulated in the original HELENA poloidal grid.
            ! loop over normal points
            do kd = 1,grid_xyz%loc_n_r                                     ! loop over all normal points
                ! loop over toroidal points
                do jd = 1,grid_xyz%n(2)
                    ! interpolate at the requested poloidal points
                    do id = 1,grid_xyz%n(1)
                        ! initialize local theta
                        theta_loc = grid_xyz%theta_E(id,jd,kd)

                        ! add or subtract 2pi to  the parallel angle until it is
                        ! at least 0 to get principal range 0..2pi
                        if (theta_loc.lt.0._dp) then
                            do while (theta_loc.lt.0._dp)
                                theta_loc = theta_loc + 2*pi
                            end do
                        else if (theta_loc.gt.2*pi) then
                            do while (theta_loc.gt.2._dp*pi)
                                theta_loc = theta_loc - 2*pi
                            end do
                        end if

                        ! take into account possible symmetry
                        if (ias.eq.0 .and. theta_loc.gt.pi) then
                            theta_loc = 2*pi-theta_loc
                            pmone = -1
                        else
                            pmone = 1
                        end if

                        ! interpolate  the HELENA  variables poloidally
                        ierr = spline(chi_h,r_h_int(:,kd),[theta_loc],&
                            &r_loc(id:id,jd,kd),ord=norm_disc_prec_eq,&
                            &bcs=bcs(:,1),bcs_val=bcs_val(:,1))            ! even
                        chckerr('')
                        ierr = spline(chi_h,pmone*z_h_int(:,kd),[theta_loc],&
                            &z(id:id,jd,kd),ord=norm_disc_prec_eq,&
                            &bcs=bcs(:,2),bcs_val=bcs_val(:,2))            ! odd
                        chckerr('')
                    end do
                end do
```

```
            end do
            if (present(r)) r = r_loc

            ! calculate X and Y, transforming cylindrical to cartesian
            ! (the geometrical zeta is the inverse of HELENA zeta)
            x = r_loc*cos(-grid_xyz%zeta_E)
            y = r_loc*sin(-grid_xyz%zeta_E)

            ! deallocate
            deallocate(r_loc)
        end function calc_xyz_grid_hel
    end function calc_xyz_grid

    !> Extend a grid angularly.
    !!
    !! This  is done  using  equidistant  variables of  \c  n_theta_plot and  \c
    !! n_zeta_plot angular and own \c loc_n_r points in F coordinates.
    !!
    !! Optionally,  the grid  can  also be  converted to  E  coordinates if  the
    !! equilibrium  grid is  provided.  This is  required  for many  operations,
    !! such  as  the  calculation  of   \f$X\f$,  \f$Y\f$  and  \f$Z\f$  through
    !! calc_XYZ_grid().
    !!
    !! \note For VMEC, it can be slow, as grid_utilities.coord_f2e() is used.
    !!
    !! \return ierr
    integer function extend_grid_f(grid_in,grid_ext,grid_eq,n_theta_plot,&
        &n_zeta_plot,lim_theta_plot,lim_zeta_plot) result(ierr)
        use num_vars, only: n_theta => n_theta_plot, n_zeta => n_zeta_plot, &
            &min_theta_plot, max_theta_plot, min_zeta_plot, max_zeta_plot
        use mpi_utilities, only: get_ser_var

        character(*), parameter :: rout_name = 'extend_grid_F'

        ! input / output
        type(grid_type), intent(in) :: grid_in                                  !< grid to be extended
        type(grid_type), intent(inout) :: grid_ext                              !< extended grid
        type(grid_type), intent(in), optional :: grid_eq                        !< equilibrium grid
        integer, intent(in), optional :: n_theta_plot                           !< number of poins in theta direction
        integer, intent(in), optional :: n_zeta_plot                            !< number of poins in zeta direction
        real(dp), intent(in), optional :: lim_theta_plot(2)                     !< limits in theta
        real(dp), intent(in), optional :: lim_zeta_plot(2)                      !< limits in zeta

        ! local variables
        type(grid_type) :: grid_ext_trim                                        ! trimmed grid_ext
        integer :: n_theta_plot_loc                                             ! local n_theta_plot
        integer :: n_zeta_plot_loc                                              ! local n_zeta_plot
        real(dp) :: lim_theta_plot_loc(2)                                       ! local limits of theta_plot
        real(dp) :: lim_zeta_plot_loc(2)                                        ! local limits of zeta_plot

        ! initialize ierr
        ierr = 0

        ! set up local variables
        n_theta_plot_loc = n_theta
        if (present(n_theta_plot)) n_theta_plot_loc = n_theta_plot
        n_zeta_plot_loc = n_zeta
        if (present(n_zeta_plot)) n_zeta_plot_loc = n_zeta_plot
        lim_theta_plot_loc = [min_theta_plot,max_theta_plot]
        if (present(lim_theta_plot)) lim_theta_plot_loc = lim_theta_plot
        lim_zeta_plot_loc = [min_zeta_plot,max_zeta_plot]
        if (present(lim_zeta_plot)) lim_zeta_plot_loc = lim_zeta_plot

        ! user ouput
        call writo('Theta limits: '//trim(r2strt(lim_theta_plot_loc(1)))//&
            &'pi .. '//trim(r2strt(lim_theta_plot_loc(2)))//'pi')
        call writo('Zeta limits: '//trim(r2strt(lim_zeta_plot_loc(1)))//&
            &'pi .. '//trim(r2strt(lim_zeta_plot_loc(2)))//'pi')

        ! creating  equilibrium  grid  for  the output  that  covers  the  whole
        ! geometry angularly in F coordinates
        ierr = grid_ext%init([n_theta_plot_loc,n_zeta_plot_loc,grid_in%n(3)],&
            &[grid_in%i_min,grid_in%i_max])
        chckerr('')
        grid_ext%r_F = grid_in%r_F
        grid_ext%loc_r_F = grid_in%loc_r_F
        ierr = calc_eqd_grid(grid_ext%theta_F,lim_theta_plot_loc(1)*pi,&
            &lim_theta_plot_loc(2)*pi,1)
        chckerr('')
        ierr = calc_eqd_grid(grid_ext%zeta_F,lim_zeta_plot_loc(1)*pi,&
            &lim_zeta_plot_loc(2)*pi,2)
        chckerr('')

        ! convert all F coordinates to E coordinates if requested
        if (present(grid_eq)) then
            call writo('convert F to E coordinates')
            call lvl_ud(1)
```

```fortran

            ! set total r_E
            grid_ext%r_E = grid_in%r_E

            ! get local r_E and angular theta_E and zeta_E
            ierr = coord_f2e(grid_eq,&
                &grid_ext%loc_r_F,grid_ext%theta_F,grid_ext%zeta_F,&
                &grid_ext%loc_r_E,grid_ext%theta_E,grid_ext%zeta_E)
            chckerr('')

            ! trim external grid
            ierr = trim_grid(grid_ext,grid_ext_trim)
            chckerr('')

            ! clean up
            call grid_ext_trim%dealloc()

            call lvl_ud(-1)
        end if
    end function extend_grid_f

    !> Copy a grid A to a new grid B, that was not yet initialized.
    !!
    !! This  new grid  can contain  a  subsection of  the previous  grid in  all
    !! dimensions. It can also be a divided grid, by providing the limits
    !!
    !! \note
    !!  -# The normal  limits for the divided  grid should be given  in terms of
    !!  the normal dimension of \c grid B.
    !!  -# if the grids are not  purely normal, the procedure can currently only
    !!  handle the copying of a \c grid_A that is not divided.
    !!
    !! \return ierr
    integer function copy_grid(grid_A,grid_B,lims_B,i_lim) result(ierr)
        character(*), parameter :: rout_name = 'copy_grid'

        ! input / output
        class(grid_type), intent(in) :: grid_a                                  !< grid to be initialized
        class(grid_type), intent(inout) :: grid_b                               !< grid to be initialized
        integer, intent(in), optional :: lims_b(3,2)                            !< ranges for subset of grid
        integer, intent(in), optional :: i_lim(2)                               !< min. and max. local normal index

        ! local variables
        integer :: n_b(3)                                                       ! n of grid B
        integer :: lims_b_loc(3,2)                                              ! local lims_B
        integer :: i_lim_loc(2)                                                 ! local i_lim

        ! initialize ierr
        ierr = 0

        ! set up dimensions and of B
        lims_b_loc(1,:) = [1,grid_a%n(1)]
        lims_b_loc(2,:) = [1,grid_a%n(2)]
        lims_b_loc(3,:) = [1,grid_a%n(3)]
        if (present(lims_b)) lims_b_loc = lims_b
        n_b = lims_b_loc(:,2)-lims_b_loc(:,1)+1
        where (lims_b_loc(:,1).eq.lims_b_loc(:,2) &
            &.and. lims_b_loc(:,1).eq.[0,0,0]) n_b = 0
        i_lim_loc = lims_b_loc(3,:)
        if (present(i_lim)) i_lim_loc = lims_b_loc(3,1) - 1 + i_lim
        ! tests
        if (n_b(1).gt.grid_a%n(1) .or. n_b(2).gt.grid_a%n(2) .or. &
            &n_b(3).gt.grid_a%n(3)) then
            write(*,*) 'n_B' ,n_b
            write(*,*) 'grid_A', grid_a%n
            ierr = 1
            chckerr('lims_B is too large')
        end if
        if (i_lim_loc(1).gt.grid_a%n(3) .or. i_lim_loc(2).gt.grid_a%n(3)) then
            ierr = 1
            chckerr('normal limits too wide')
        end if

        ! allocate the new grid
        ierr = grid_b%init(n_b,i_lim)
        chckerr('')

        ! copy arrays, possibly subset
        grid_b%r_F = grid_a%r_F(lims_b_loc(3,1):lims_b_loc(3,2))
        grid_b%r_E = grid_a%r_E(lims_b_loc(3,1):lims_b_loc(3,2))
        grid_b%loc_r_F = grid_a%r_F(i_lim_loc(1):i_lim_loc(2))
        grid_b%loc_r_E = grid_a%r_E(i_lim_loc(1):i_lim_loc(2))
        if (n_b(1).ne.0 .and. n_b(2).ne.0) then
            if (grid_a%divided) then
                ierr = 1
                chckerr('grid_A cannot be divided')
            end if
```

```
            grid_b%theta_F = grid_a%theta_F(lims_b_loc(1,1):lims_b_loc(1,2),&
                &lims_b_loc(2,1):lims_b_loc(2,2),i_lim_loc(1):i_lim_loc(2))
            grid_b%theta_E = grid_a%theta_E(lims_b_loc(1,1):lims_b_loc(1,2),&
                &lims_b_loc(2,1):lims_b_loc(2,2),i_lim_loc(1):i_lim_loc(2))
            grid_b%zeta_F = grid_a%zeta_F(lims_b_loc(1,1):lims_b_loc(1,2),&
                &lims_b_loc(2,1):lims_b_loc(2,2),i_lim_loc(1):i_lim_loc(2))
            grid_b%zeta_E = grid_a%zeta_E(lims_b_loc(1,1):lims_b_loc(1,2),&
                &lims_b_loc(2,1):lims_b_loc(2,2),i_lim_loc(1):i_lim_loc(2))
        end if
    end function copy_grid

    !> Calculates volume integral on a 3D grid.
    !!
    !! Two angular and  one normal variable have  to be provided on a the grid.
    !!
    !! If the <tt>i</tt>'th dimension of the  grid is equal to one, the function
    !! to be integrated is assumed not to vary in this dimension.
    !!
    !! Furthermore, if \c i is 1 or 2, the corresponding <tt>i</tt>'th (angular)
    !! variable is the only variable that  is assumed to vary in that dimension.
    !! The other angular variable as well  as the normal variable are assumed to
    !! be constant like the function itself,  resulting in a factor \f$2 \pi\f$.
    !! However, if \c i is 3, an error is displayed as this does not represent a
    !! physical situation.
    !!
    !! A common case  through which to understand this is  the axisymmetric case
    !! where the first angular variable  \f$\theta\f$ varies in the dimensions 1
    !! and  3,  the second  angular  variable  \f$\zeta\f$  varies only  in  the
    !! dimension 2, and the normal variable only varies in the dimension 3.
    !!
    !! Alternatively, there  is the  case of a  grid-aligned set  of coordinates
    !! \f$\theta\f$ and  \f$\alpha\f$, where the first  dimension corresponds to
    !! the direction along the magnetic field line, the second to the geodesical
    !! direction and the third to the  normal direction. If the calculations for
    !! different  field  lines  are  decoupled,  the  variation  in  the  second
    !! dimension is not taken into account and no integration happens along it.
    !!
    !! Internally, the angular variables and  the normal variable are related to
    !! the  coordinates \f$\left(x,y,z\right)\f$  that correspond  to the  three
    !! dimensions. They thus  form a computational orthogonal grid  to which the
    !! original coordinates are related through the transformation of Jacobians:
    !!  \f[\mathcal{J}_{xyz} = \mathcal{J}_\text{F}
    !!      \frac{\partial r_\text{F}}{\partial z}
    !!      \left(\frac{\partial \gamma_1}{\partial x}
    !!      \frac{\gamma_2}{\partial y}-
    !!      \frac{\partial \gamma_1}{\partial y}
    !!      \frac{\gamma_2}{\partial x}\right) , \f]
    !! where \f$\gamma_1\f$  and \f$\gamma_2\f$  are \c ang_1  and \c  ang_2, so
    !! that the integral becomes
    !!  \f[ \sum_{x,y,z} \left[ f\left(x,y,z\right) \mathcal{J}_\text{F}
    !!      \frac{\partial r_\text{F}}{\partial z}
    !!      \left(\frac{\partial \gamma_1}{\partial x}
    !!      \frac{\gamma_2}{\partial y}-
    !!      \frac{\partial \gamma_1}{\partial y}
    !!      \frac{\gamma_2}{\partial x}\right)
    !!      \Delta_x \Delta_y \Delta_z \right] \f]
    !! where \f$\Delta_x\f$, \f$\Delta_y\f$ and \f$\Delta_z\f$ are all trivially
    !! equal to 1.
    !!
    !! The integrand  has to be  evaluated at the intermediate  positions inside
    !! the cells. This  is done by taking the average  of the \f$2^3=8\f$ points
    !! for  \c fj  = \f$f  J_\text{F}\f$ as  well as  the transformation  of the
    !! Jacobian to \f$\left(x,y,z\right)\f$ coordinates.
    !!
    !! \see See grid_vars.grid_type for a discussion on \c ang_1 and \c ang_2.
    !!
    !! \note
    !!  -# if the coordinates are independent,  this method is equivalent to the
    !!  repeated  numerical integration  using  the trapezoidal  method, \b  NOT
    !!  Simpson's 3/8 rule!
    !!  -# by setting \c debug_calc_int_vol, this  method can be compared to the
    !!  trapezoidal and simple method for  independent coordinates, again \b NOT
    !!  for Simpson's 3/8 rule!
    !!  -# The  Simpson's 3/8  rule could be  developed but it  is not  of great
    !!  importance.
    !!
    !! \return ierr
    integer function calc_int_vol(ang_1,ang_2,norm,J,f,f_int) result(ierr)
#if ldebug
        use num_vars, only: rank, n_procs
#endif

        character(*), parameter :: rout_name = 'calc_int_vol'

        ! input / output
        real(dp), intent(in) :: ang_1(:,:,:)                                    !< coordinate variable \f$\gamma_1\f$
        real(dp), intent(in) :: ang_2(:,:,:)                                    !< coordinate variable \f$\gamma_2\f$
```

```
        real(dp), intent(in) :: norm(:)                                     !< coordinate variable \f$r_\text{F}\f$
        real(dp), intent(in) :: j(:,:,:)                                    !< Jacobian
        complex(dp), intent(in) :: f(:,:,:,:)                               !< input f(n_par,n_geo,n_r,size_X^2)
        complex(dp), intent(inout) :: f_int(:)                              !< output integrated f

        ! local variables
        integer :: id, jd, kd, ld                                           ! counters
        integer :: nn_mod                                                   ! number of indices for V and V_int
        integer :: k_min                                                    ! minimum k
        integer :: dims(3)                                                  ! real dimensions
        real(dp), allocatable :: transf_j(:,:,:,:)                          ! comps. of transf. between J and Jxyz: theta_x, zeta_y, theta_y, zeta_x and r_F_z
        real(dp), allocatable :: transf_j_tot(:,:,:)                        ! transf. between J and Jxyz: (theta_x*zeta_y-theta_y*zeta_x)*r_F_z
        complex(dp), allocatable :: jf(:,:,:,:)                             ! J*f
        character(len=max_str_ln) :: err_msg                                ! error message
        integer :: i_a(3), i_b(3)                                           ! limits of building blocks of intermediary variables
        logical :: dim_1(2)                                                 ! whether the dimension is equal to one
#if ldebug
        character(len=max_str_ln), allocatable :: var_names(:,:)            ! names of variables
        complex(dp), allocatable :: f_int_alt(:)                            ! alternative calculation for output
        complex(dp), allocatable :: f_int_alt_1d(:,:)                       ! intermediary step in f_int_ALT
        complex(dp), allocatable :: f_int_alt_2d(:,:,:)                     ! intermediary step in f_int_ALT
        complex(dp), allocatable :: f_int_alt_alt(:)                        ! another alternative calculation for output
        integer :: loc_norm(2)                                              ! local normal index
#endif

        ! initialize ierr
        ierr = 0

        ! set nn_mod
        nn_mod = size(f,4)

        ! tests
        if (size(f_int).ne.nn_mod) then
            ierr = 1
            err_msg = 'f and f_int need to have the same storage convention'
            chckerr(err_msg)
        end if
        if (size(ang_1).ne.size(ang_2) .or. size(ang_1,3).ne.size(norm)) then
            ierr = 1
            err_msg = 'The angular variables have to be compatible'
            chckerr(err_msg)
        end if
        if (size(norm).eq.1) then
            ierr = 1
            err_msg = 'The normal grid size has to be greater than one'
            chckerr(err_msg)
        end if

        ! set up dims
        dims = shape(ang_1)

        ! set up dim_1
        dim_1 = dims(1:2).eq.1

        ! set up Jf and transf_J
        allocate(jf(max(dims(1)-1,1),max(dims(2)-1,1),dims(3)-1,nn_mod))
        allocate(transf_j(max(dims(1)-1,1),max(dims(2)-1,1),dims(3)-1,5))
        allocate(transf_j_tot(max(dims(1)-1,1),max(dims(2)-1,1),dims(3)-1))

        ! set up k_min
        k_min = 1

        ! get intermediate Jf and components of transf_J:
        !   1: ang_1_x
        !   2: ang_2_y
        !   3: ang_1_y
        !   4: ang_2_x
        !   5: r_F_z
        ! Note: the are always 8 terms in the summation, hence the factor 0.125
        jf = 0._dp
        transf_j = 0._dp
        do kd = 1,2
            if (kd.eq.1) then
                i_a(3) = 1
                i_b(3) = dims(3)-1
            else
                i_a(3) = 2
                i_b(3) = dims(3)
            end if
            do jd = 1,2
                if (dim_1(2)) then
                    i_a(2) = 1
                    i_b(2) = dims(2)
                else
```

```
                    if (jd.eq.1) then
                        i_a(2) = 1
                        i_b(2) = dims(2)-1
                    else
                        i_a(2) = 2
                        i_b(2) = dims(2)
                    end if
                end if
                do id = 1,2
                    if (dim_1(1)) then
                        i_a(1) = 1
                        i_b(1) = dims(1)
                    else
                        if (id.eq.1) then
                            i_a(1) = 1
                            i_b(1) = dims(1)-1
                        else
                            i_a(1) = 2
                            i_b(1) = dims(1)
                        end if
                    end if
                    do ld = 1,nn_mod
                        ! Jf
                        jf(:,:,:,ld) = jf(:,:,:,ld) + 0.125_dp * &
                            &j(i_a(1):i_b(1),i_a(2):i_b(2),i_a(3):i_b(3)) * &
                            &f(i_a(1):i_b(1),i_a(2):i_b(2),i_a(3):i_b(3),ld)
                    end do
                    if (dim_1(1)) then
                        ! transf_J(1): ang_1_x
                        transf_j(:,:,:,1) = 2*pi
                        ! transf_J(4): ang_2_x
                        transf_j(:,:,:,4) = 0._dp
                    else
                        do ld = 1,dims(1)-1
                            ! transf_J(1): ang_1_x
                            transf_j(ld,:,:,1) = transf_j(ld,:,:,1) + &
                                &0.125_dp * ( &
                                &ang_1(ld+1,i_a(2):i_b(2),i_a(3):i_b(3)) - &
                                &ang_1(ld,i_a(2):i_b(2),i_a(3):i_b(3)) )
                            ! transf_J(4): ang_2_x
                            transf_j(ld,:,:,4) = transf_j(ld,:,:,4) + &
                                &0.125_dp * ( &
                                &ang_2(ld+1,i_a(2):i_b(2),i_a(3):i_b(3)) - &
                                &ang_2(ld,i_a(2):i_b(2),i_a(3):i_b(3)) )
                        end do
                    end if
                    if (dim_1(2)) then
                        ! transf_J(2): ang_2_y
                        transf_j(:,:,:,2) = 2*pi
                        ! transf_J(3): ang_1_y
                        transf_j(:,:,:,3) = 0._dp
                    else
                        do ld = 1,dims(2)-1
                            ! transf_J(2): ang_2_y
                            transf_j(:,ld,:,2) = transf_j(:,ld,:,2) + &
                                &0.125_dp * ( &
                                &ang_2(i_a(1):i_b(1),ld+1,i_a(3):i_b(3)) - &
                                &ang_2(i_a(1):i_b(1),ld,i_a(3):i_b(3)) )
                            ! transf_J(3): ang_1_y
                            transf_j(:,ld,:,3) = transf_j(:,ld,:,3) + &
                                &0.125_dp * ( &
                                &ang_1(i_a(1):i_b(1),ld+1,i_a(3):i_b(3)) - &
                                &ang_1(i_a(1):i_b(1),ld,i_a(3):i_b(3)) )
                        end do
                    end if
                    do ld = 1,dims(3)-1
                        ! transf_J(5): r_F_z
                        transf_j(:,:,ld,5) = transf_j(:,:,ld,5) + 0.125_dp * ( &
                            &norm(ld+1) - norm(ld) )
                    end do
                end do
            end do
        end do

        ! set up total transf_J
        transf_j_tot = (transf_j(:,:,:,1)*transf_j(:,:,:,2)-&
            &transf_j(:,:,:,3)*transf_j(:,:,:,4))*transf_j(:,:,:,5)

        ! integrate term  over ang_par_F for all equilibrium  grid points, using
        ! dx = dy = dz = 1
        do ld = 1,nn_mod
            f_int(ld) = sum(jf(:,:,:,ld)*transf_j_tot)
        end do

#if ldebug
        if (debug_calc_int_vol) then
            call writo('Testing whether volume integral is calculated &
```

```
                &correctly')
            call lvl_ud(1)

            allocate(var_names(nn_mod,2))
            var_names(:,1) = 'real part of integrand J f'
            var_names(:,2) = 'imaginary part of integrand J f'
            do ld = 1,nn_mod
                var_names(ld,1) = trim(var_names(ld,1))//' '//trim(i2str(ld))
                var_names(ld,2) = trim(var_names(ld,2))//' '//trim(i2str(ld))
            end do

            call plot_hdf5(var_names(:,1),'TEST_RE_Jf_'//trim(i2str(rank)),&
                &rp(jf))
            call plot_hdf5(var_names(:,2),'TEST_IM_Jf_'//trim(i2str(rank)),&
                &ip(jf))
            call plot_hdf5('transformation of Jacobians',&
                &'TEST_transf_J_'//trim(i2str(rank)),transf_j_tot)

            ! do alternative calculations
            ! assuming that coordinate i varies only in dimensions i!!!
            allocate(f_int_alt(nn_mod))
            allocate(f_int_alt_1d(size(f,3),nn_mod))
            allocate(f_int_alt_2d(size(f,2),size(f,3),nn_mod))
            allocate(f_int_alt_alt(nn_mod))
            f_int_alt = 0._dp
            f_int_alt_1d = 0._dp
            f_int_alt_2d = 0._dp
            f_int_alt_alt = 0._dp

            ! integrate in first coordinate
            do kd = 1,size(f,3)
                do jd = 1,size(f,2)
                    if (size(f,1).gt.1) then
                        do id = 2,size(f,1)
                            f_int_alt_2d(jd,kd,:) = f_int_alt_2d(jd,kd,:) + &
                                &0.5*(j(id,jd,kd)*f(id,jd,kd,:)+&
                                &j(id-1,jd,kd)*f(id-1,jd,kd,:))*&
                                &(ang_1(id,jd,kd)-ang_1(id-1,jd,kd))
                        end do
                    else
                        f_int_alt_2d(jd,kd,:) = j(1,jd,kd)*f(1,jd,kd,:)
                    end if
                end do
            end do

            ! integrate in second coordinate
            do kd = 1,size(f,3)
                if (size(f,2).gt.1) then
                    do jd = 2,size(f,2)
                        f_int_alt_1d(kd,:) = f_int_alt_1d(kd,:) + &
                            &0.5*(f_int_alt_2d(jd,kd,:)+&
                            &f_int_alt_2d(jd-1,kd,:))*&
                            &(ang_2(1,jd,kd)-ang_2(1,jd-1,kd))                  ! assuming that ang_2 is not dependent on dimension 1
                    end do
                else
                    f_int_alt_1d(kd,:) = f_int_alt_2d(1,kd,:)
                end if
            end do

            ! integrate in third coordinate
            if (size(f,3).gt.1) then
                do kd = 2,size(f,3)
                    f_int_alt(:) = f_int_alt(:) + &
                        &0.5*(f_int_alt_1d(kd,:)+f_int_alt_1d(kd-1,:))*&
                        &(norm(kd)-norm(kd-1))
                end do
            else
                f_int_alt = f_int_alt_1d(1,:)
            end if

            ! second alternative, first order integration
            loc_norm = [1,size(f,3)]
            if (rank.lt.n_procs-1) loc_norm(2) = loc_norm(2)-1                  ! no ghost region needed for this method
            do ld = 1,size(f,4)
                f_int_alt_alt(ld) = sum(j(:,:,loc_norm(1):loc_norm(2))*&
                    &f(:,:,loc_norm(1):loc_norm(2),ld))
            end do
            if (size(f,1).gt.1) f_int_alt_alt = f_int_alt_alt*&
                &(ang_1(2,1,1)-ang_1(1,1,1))
            if (size(f,2).gt.1) f_int_alt_alt = f_int_alt_alt*&
                &(ang_2(1,2,1)-ang_2(1,1,1))
            if (size(f,3).gt.1) f_int_alt_alt = f_int_alt_alt*&
                &(norm(2)-norm(1))

            call writo('Resulting integral: ',persistent=.true.)
            call lvl_ud(1)
```

```
            do ld = 1,nn_mod
                call writo('rank '//trim(i2str(rank))//' integral      '//&
                    &trim(i2str(ld))//': '//trim(c2str(f_int(ld))),&
                    &persistent=.true.)
                call writo('rank '//trim(i2str(rank))//'   alternative '//&
                    &trim(i2str(ld))//': '//trim(c2str(f_int_alt(ld))),&
                    &persistent=.true.)
                call writo('rank '//trim(i2str(rank))//'   other alt.  '//&
                    &trim(i2str(ld))//': '//trim(c2str(f_int_alt_alt(ld))),&
                    &persistent=.true.)
            end do
            call lvl_ud(-1)

            call writo('(Note that alternative calculations are only valid if &
                &independent coordinates!)')

            call lvl_ud(-1)
        end if
#endif

        ! deallocate local variables
        deallocate(jf,transf_j,transf_j_tot)
    end function calc_int_vol

    !> Trim a grid, removing any overlap between the different regions.
    !!
    !! by default, the routine assumes a symmetric ghost region and cuts as many
    !! grid points from the end of the previous process as from the beginning of
    !! the next  process, but if the  number of overlapping grid  points is odd,
    !! the next process looses one more point.
    !!
    !! optionally, the trimmed  indices in the normal direction  can be provided
    !! in  \c  norm_id,  i.e.  the  indices in  the  old,  untrimmed  grid  that
    !! correspond to the start and end indices of the trimmed grid. E.g. if
    !!  - proc 0:  3 ... 25
    !!  - proc 1: 20 ... 50
    !!
    !! then the trimmed grid will be:
    !!  - proc 0:  3 ... 22
    !!  - proc 1: 23 ... 50
    !!
    !! which is shifted down by 2 to
    !!  - proc 0:  1 ... 20
    !!  - proc 1: 21 ... 48
    !!
    !! in the trimmed  grid. The indices of the  previous step (3 & 22  and 23 &
    !! 50) are saved in \c norm_id.
    !!
    !! \return ierr
    integer function trim_grid(grid_in,grid_out,norm_id) result(ierr)
        use num_vars, only: n_procs, rank
        use mpi_utilities, only: get_ser_var

        character(*), parameter :: rout_name = 'trim_grid'

        ! input / output
        type(grid_type), intent(in) :: grid_in                                  !< input grid
        type(grid_type), intent(inout) :: grid_out                              !< trimmed grid
        integer, intent(inout), optional :: norm_id(2)                          !< normal indices corresponding to trimmed part

        ! local variables
        integer, allocatable :: tot_i_min(:)                                    ! i_min of grid of all processes
        integer, allocatable :: tot_i_max(:)                                    ! i_max of grid of all processes
        integer :: i_lim_out(2)                                                 ! i_lim of output grid
        integer :: n_out(3)                                                     ! n of output grid

        ! initialize ierr
        ierr = 0

        ! detect whether grid divided
        if (grid_in%divided) then
            ! get min_i's of the grid_in
            ierr = get_ser_var([grid_in%i_min],tot_i_min,scatter=.true.)
            chckerr('')

            ! get max_i's of the grid_in
            ierr = get_ser_var([grid_in%i_max],tot_i_max,scatter=.true.)
            chckerr('')

            ! set  i_lim of trimmed output  grid (not yet shifted  by first proc
            ! min)
            if (rank.gt.0) then
                i_lim_out(1) = max(tot_i_min(1),tot_i_min(rank+1)+&
                    &ceiling((tot_i_max(rank)-tot_i_min(rank+1)+1._dp)/2))
            else
                i_lim_out(1) = tot_i_min(1)
```

```
            end if
            if (rank.lt.n_procs-1) then
                i_lim_out(2) = min(tot_i_max(n_procs),tot_i_max(rank+1)-&
                    &floor((tot_i_max(rank+1)-tot_i_min(rank+2)+1._dp)/2))
            else
                i_lim_out(2) = tot_i_max(n_procs)
            end if

            ! limit to input limits (necessary if a process has only one point)
            i_lim_out(1) = max(min(i_lim_out(1),grid_in%i_max),grid_in%i_min)
            i_lim_out(2) = max(min(i_lim_out(2),grid_in%i_max),grid_in%i_min)

            ! get  min_i's and max_i's  of the grid_out,  not shifted by  min of
            ! first process
            ierr = get_ser_var([i_lim_out(1)],tot_i_min,scatter=.true.)
            chckerr('')
            ierr = get_ser_var([i_lim_out(2)],tot_i_max,scatter=.true.)
            chckerr('')

            ! set n of output grid
            n_out(1) = grid_in%n(1)
            n_out(2) = grid_in%n(2)
            n_out(3) = tot_i_max(n_procs)-tot_i_min(1)+1

            ! create new grid
            ierr = grid_out%init(n_out,i_lim_out-tot_i_min(1)+1,divided=.true.) ! limits shifted by min of first process
            chckerr('')

            ! recycle  i_lim_out  for  shifted  array  indices, set  norm_id  if
            ! requested
            i_lim_out = i_lim_out - grid_in%i_min + 1
            if (present(norm_id)) norm_id = i_lim_out
        else
            ! set n of output grid
            n_out = grid_in%n

            ! set unshifted i_lim_out and norm_id if requested
            i_lim_out = [grid_in%i_min,grid_in%i_max]
            if (present(norm_id)) norm_id = i_lim_out

            ! shift i_lim_out by min.
            i_lim_out = i_lim_out-i_lim_out(1)+1

            ! create new grid
            ierr = grid_out%init(n_out,i_lim_out)                               ! grid not divided
            chckerr('')
        end if

        ! copy local arrays
        if (grid_in%n(1).ne.0 .and. grid_in%n(2).ne.0) then                     ! only if 3D grid
            grid_out%theta_E = grid_in%theta_E(:,:,i_lim_out(1):i_lim_out(2))
            grid_out%zeta_E = grid_in%zeta_E(:,:,i_lim_out(1):i_lim_out(2))
            grid_out%theta_F = grid_in%theta_F(:,:,i_lim_out(1):i_lim_out(2))
            grid_out%zeta_F = grid_in%zeta_F(:,:,i_lim_out(1):i_lim_out(2))
        end if
        if (grid_in%divided) then                                               ! but if input grid divided, loc_r gets
    priority
            grid_out%loc_r_E = grid_in%loc_r_E(i_lim_out(1):i_lim_out(2))
            grid_out%loc_r_F = grid_in%loc_r_F(i_lim_out(1):i_lim_out(2))
        end if

        ! if divided, set total arrays
        if (grid_in%divided) then
            grid_out%r_E = grid_in%r_E(tot_i_min(1):tot_i_max(n_procs))
            grid_out%r_F = grid_in%r_F(tot_i_min(1):tot_i_max(n_procs))
        else
            grid_out%r_E = grid_in%r_E
            grid_out%r_F = grid_in%r_F
        end if

        ! if allocated, copy trigonometric factors
        if (allocated(grid_in%trigon_factors)) then
            allocate(grid_out%trigon_factors(&
                &size(grid_in%trigon_factors,1),&
                &size(grid_in%trigon_factors,2),&
                &size(grid_in%trigon_factors,3),&
                &grid_out%loc_n_r,2))
            grid_out%trigon_factors = &
                &grid_in%trigon_factors(:,:,:,i_lim_out(1):i_lim_out(2),:)
        end if
    end function trim_grid

    !> Untrims a trimmed grid by  introducing an asymmetric ghost regions at the
    !! right.
    !!
    !! The width of the ghost region has to be provided.
    !!
```

```
    !! \note
    !!  -# the ghosted grid should be deallocated (with dealloc_grid()).
    !!  -# the input grid has to be trimmed.
    !!
    !! \return ierr
    integer function untrim_grid(grid_in,grid_out,size_ghost) result(ierr)
        use num_vars, only: n_procs, rank
        use mpi_utilities, only: get_ghost_arr, get_ser_var

        character(*), parameter :: rout_name = 'untrim_grid'

        ! input / output
        type(grid_type), intent(in) :: grid_in                                  !< input grid
        type(grid_type), intent(inout) :: grid_out                              !< ghosted grid
        integer, intent(in) :: size_ghost                                       !< width of ghost region

        ! local variables
        integer :: i_lim_in(2)                                                  ! limits of input grid
        integer :: i_lim_out(2)                                                 ! limits of ghosted grid
        integer, allocatable :: tot_loc_n_r(:)                                  ! loc_n_r of all processes
        integer :: size_ghost_loc                                               ! local size_ghost

        ! initialize ierr
        ierr = 0

        ! get array sizes of all processes
        ierr = get_ser_var([grid_in%loc_n_r],tot_loc_n_r,scatter=.true.)
        chckerr('')

        ! set local size_ghost
        size_ghost_loc = min(size_ghost,minval(tot_loc_n_r)-1)

        ! set i_lim_in and i_lim_out
        i_lim_in = [grid_in%i_min,grid_in%i_max]
        i_lim_out = i_lim_in
        if (rank.lt.n_procs-1) i_lim_out(2) = i_lim_out(2)+size_ghost_loc

        ! create grid
        ierr = grid_out%init(grid_in%n,i_lim_out)
        chckerr('')

        ! set ghosted variables
        if (grid_in%n(1).ne.0 .and. grid_in%n(2).ne.0) then                     ! only if 3d grid
            grid_out%theta_e(:,:,1:grid_in%loc_n_r) = grid_in%theta_e
            grid_out%zeta_e(:,:,1:grid_in%loc_n_r) = grid_in%zeta_e
            grid_out%theta_f(:,:,1:grid_in%loc_n_r) = grid_in%theta_f
            grid_out%zeta_f(:,:,1:grid_in%loc_n_r) = grid_in%zeta_f
            ierr = get_ghost_arr(grid_out%theta_e,size_ghost_loc)
            chckerr('')
            ierr = get_ghost_arr(grid_out%zeta_e,size_ghost_loc)
            chckerr('')
            ierr = get_ghost_arr(grid_out%theta_f,size_ghost_loc)
            chckerr('')
            ierr = get_ghost_arr(grid_out%zeta_f,size_ghost_loc)
            chckerr('')
        end if
        if (grid_in%divided) then                                               ! but if input grid divided, loc_r gets priority
            grid_out%loc_r_e = grid_in%r_e(i_lim_out(1):i_lim_out(2))
            grid_out%loc_r_f = grid_in%r_f(i_lim_out(1):i_lim_out(2))
        end if
        grid_out%r_e = grid_in%r_e
        grid_out%r_f = grid_in%r_f
    end function untrim_grid

    !> Calculates  contra- and  covariant  components of  a  vector in  multiple
    !! coordinate systems.
    !!
    !! Chain of coordinate systems considered:
    !!  -# Flux F: \f$(\alpha,\psi,\theta)\f$,
    !!  -# Magnetic M: \f$(\psi,\theta,\zeta)\f$,
    !!      - H: \f$(\psi,\theta,\phi)\f$ if HELENA is used,
    !!      - V: \f$(r,\theta,\zeta)\f$ if VMEC is used,
    !!  -# Cylindrical C: \f$(R,\varphi,Z)\f$,
    !!  -# Cartesian X: \f$(X,Y,Z)\f$,
    !! as well  as the magnitude, starting  from the input in  Flux coordinates.
    !! Note that  the Cartesian components  in X,  Y and Z  can be used  to plot
    !! the  real  vector  and  that  covariant components  should  be  equal  to
    !! contravariant components in this coordinate system.
    !!
    !! Also, the fluxes can be calculated and plot by providing a \c base_name.
    !!
    !! By  default, the  cartesian  components  are returned,  but  this can  be
    !! indicated differently by providing \c max_transf.
    !!
    !! \note
    !!  -# Plots  for different Richardson  levels can  be combined to  show the
```

```
    !!  total grid by just plotting them all individually and sequentially.
    !!  -# The metric factors and  transformation matrices have to be allocated.
    !!  They can be calculated using the routines from eq_ops, for
    !!  <tt>deriv = [0,0,0]</tt>.
    !!  -# For VMEC, the trigonometric factors of \c grid_XYZ must be calculated
    !!  beforehand.  Optionally, by  providing \c  X, \c  Y and  \c Z,  the ones
    !!  calculated  in this  routine are  overwritten. This  is useful  for, for
    !!  example, slab geometries.
    !!  -# The  normalization factors are taken  into account and the  output is
    !!  transformed back to unnormalized values.
    !!
    !! \return ierr
    integer function calc_vec_comp(grid,grid_eq,eq_1,eq_2,v_com,norm_disc_prec,&
        &v_mag,base_name,max_transf,v_flux_tor,v_flux_pol,XYZ,compare_tor_pos) &
        &result(ierr)

        use eq_vars, only: eq_1_type, eq_2_type, max_flux_f
        use eq_utilities, only: calc_inv_met
        use num_vars, only: eq_jobs_lims, eq_job_nr, use_pol_flux_f, eq_style, &
            &use_normalization, rank, tol_zero, rz_0, &
            &compare_tor_pos_glob => compare_tor_pos
        use num_utilities, only: c, calc_int, order_per_fun, spline
        use eq_vars, only: r_0, b_0, psi_0
        use vmec_utilities, only: calc_trigon_factors
        use mpi_utilities, only: get_ser_var

        character(*), parameter :: rout_name = 'calc_vec_comp'

        ! input / output
        type(grid_type), intent(in) :: grid                                     !< grid on which vector is calculated
        type(grid_type), intent(in) :: grid_eq                                  !< grid on which equilibrium variables are calculated
        type(eq_1_type), intent(in) :: eq_1                                     !< flux equilibrium quantities
        type(eq_2_type), intent(in) :: eq_2                                     !< metric equilibrium variables
        real(dp), intent(inout) :: v_com(:,:,:,:,:)                             !< covariant and contravariant components <tt>(dim1,dim2,dim3,3,2)</tt>
        integer, intent(in) :: norm_disc_prec                                   !< precision for normal derivatives
        real(dp), intent(inout), optional :: v_mag(:,:,:)                       !< magnitude <tt>(dim1,dim2,dim3)</tt>
        character(len=*), intent(in), optional :: base_name                     !< base name for output plotting
        integer, intent(in), optional :: max_transf                             !< maximum transformation level (2: Magnetic, 3: Equilibrium, 4: Cylindrical, 5: Cartesian [def])
        real(dp), intent(inout), allocatable, optional :: v_flux_pol(:,:)       !< poloidal flux as function of normal coordinate for all toroidal positions
        real(dp), intent(inout), allocatable, optional :: v_flux_tor(:,:)       !< toroidal flux as function of normal coordinate for all poloidal positions
        real(dp), intent(in), optional :: xyz(:,:,:,:)                          !< \f$X\f$, \f$Y\f$ and \f$Z\f$ of grid
        logical, intent(in), optional :: compare_tor_pos                        !< compare toroidal positions

        ! local variables
        type(grid_type) :: grid_trim                                            ! trimmed plot grid
        integer :: max_transf_loc                                               ! local max_transf
        integer :: id, jd, kd, ld                                               ! counters
        integer :: plot_dim(4)                                                  ! dimensions of plot
        integer :: plot_offset(4)                                               ! local offset of plot
        integer :: c_loc                                                        ! local c
        integer :: tor_id(2)                                                    ! toroidal indices
        integer :: norm_id(2)                                                   ! untrimmed normal indices for trimmed grids
        logical :: cont_plot                                                    ! continued plot
        logical :: do_plot                                                      ! perform plotting
        logical :: compare_tor_pos_loc                                          ! local compare_tor_pos
        character(len=max_str_ln) :: description(3)                             ! description of plots
        character(len=max_str_ln) :: file_names(3)                              ! plot file names
        character(len=max_str_ln) :: var_names(3,2)                             ! variable names
        character(len=5) :: coord_names(3)                                      ! name of coordinates
        character(len=max_str_ln) :: err_msg                                    ! error message
        real(dp) :: norm_len                                                    ! normalization factor for lengths, to cancel the one introduced in "calc_XYZ_grid"
        real(dp), allocatable :: q_saf(:,:)                                     ! interpolated q_saf_FD and derivative
        real(dp), allocatable :: jac(:,:,:)                                     ! interpolated jac_FD
        real(dp), allocatable :: xyzr(:,:,:,:)                                  ! X, Y, Z and R of surface in cylindrical coordinates, untrimmed grid
        real(dp), allocatable :: theta_geo(:,:,:)                               ! geometrical theta
        real(dp), allocatable :: x(:,:,:,:), y(:,:,:,:), z(:,:,:,:)             ! copy of X, Y and Z, trimmed grid
        real(dp), allocatable :: v_temp(:,:,:,:,:)                              ! temporary variable for v
        real(dp), allocatable :: v_ser_temp(:)                                  ! temporary serial variable
        real(dp), allocatable :: v_ser_temp_int(:)                              ! temporary integrated serial variable
        real(dp), allocatable :: t_ba(:,:,:,:,:,:,:), t_ab(:,:,:,:,:,:,:)       ! transformation matrices from A to B
        real(dp), allocatable :: d1r(:,:,:), d2r(:,:,:)                         ! dR/dpsi, dR/dtheta in HELENA coords.
        real(dp), allocatable :: d1z(:,:,:), d2z(:,:,:)                         ! dZ/dpsi, dZ/dtheta in HELENA coords.

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Prepare calculation of vector components')
        call lvl_ud(1)

```

```
        ! set up maximum level up to which to transform and whether to plot
        max_transf_loc = 5
        if (present(max_transf)) then
            if (max_transf.ge.1 .and. max_transf.le.5) &
                &max_transf_loc = max_transf
        end if
        do_plot = .false.
        if (present(base_name)) then
            if (trim(base_name).ne.'') do_plot = .true.
        end if

        ! set up continued plot
        cont_plot = eq_job_nr.gt.1

        ! set up normalization factor
        norm_len = 1._dp
        if (use_normalization) norm_len = r_0

        ! trim plot grid
        ierr = trim_grid(grid,grid_trim,norm_id)
        chckerr('')

        ! set up X, Y, Z and R in grid
        allocate(xyzr(grid%n(1),grid%n(2),grid%loc_n_r,4))
        ierr = calc_xyz_grid(grid_eq,grid,xyzr(:,:,:,1),xyzr(:,:,:,2),&
            &xyzr(:,:,:,3),r=xyzr(:,:,:,4))
        chckerr('')

        ! set up local compare_tor_pos
        compare_tor_pos_loc = compare_tor_pos_glob
        if (present(compare_tor_pos)) compare_tor_pos_loc = compare_tor_pos

        ! get geometrical poloidal angle if comparing toroidal position
        if (compare_tor_pos_loc) then
            allocate(theta_geo(grid%n(1),grid%n(2),grid%loc_n_r))
            theta_geo = atan2(xyzr(:,:,:,3)-rz_0(2),xyzr(:,:,:,4)-rz_0(1))
            where (theta_geo.lt.0) theta_geo = theta_geo + 2*pi
        end if

        ! if plotting is required
        if (do_plot) then
            ! set up plot dimensions and local dimensions
            plot_dim = [grid_trim%n,3]
            plot_offset = [0,0,grid_trim%i_min-1,0]
            tor_id = [1,size(v_com,2)]
            if (compare_tor_pos_loc) then
                if (plot_dim(2).ne.3) then
                    ierr = 1
                    err_msg = 'When comparing toroidal positions, need to &
                        &have 3 toroidal points (one in the middle)'
                    chckerr(err_msg)
                end if
                plot_dim(2) = 1
                tor_id = [2,2]
            end if

            ! possibly modify if multiple equilibrium parallel jobs
            if (size(eq_jobs_lims,2).gt.1) then
                plot_dim(1) = eq_jobs_lims(2,size(eq_jobs_lims,2)) - &
                    &eq_jobs_lims(1,1) + 1
                plot_offset(1) = eq_jobs_lims(1,eq_job_nr) - 1
                if (compare_tor_pos_loc) then
                    ierr = 1
                    err_msg = 'When comparing toroidal positions, cannot have &
                        &multiple equilibrium jobs'
                    chckerr(err_msg)
                end if
            end if

            ! tests
            if (eq_style.eq.1 .and. .not.allocated(grid%trigon_factors)) then
                ierr = 1
                err_msg = 'trigonometric factors not allocated'
                chckerr(err_msg)
            end if

            ! user output
            if (compare_tor_pos_loc) call writo('Comparing toroidal positions')

            ! copy X, Y and Z to trimmed grid copies
            allocate(x(grid_trim%n(1),grid_trim%n(2),grid_trim%loc_n_r,1))
            allocate(y(grid_trim%n(1),grid_trim%n(2),grid_trim%loc_n_r,1))
            allocate(z(grid_trim%n(1),grid_trim%n(2),grid_trim%loc_n_r,1))
            if (present(xyz)) then
                x(:,:,:,1) = xyz(:,:,norm_id(1):norm_id(2),1)
                y(:,:,:,1) = xyz(:,:,norm_id(1):norm_id(2),2)
                z(:,:,:,1) = xyz(:,:,norm_id(1):norm_id(2),3)
```

```
            else
                x(:,:,:,1) = xyzr(:,:,norm_id(1):norm_id(2),1)
                y(:,:,:,1) = xyzr(:,:,norm_id(1):norm_id(2),2)
                z(:,:,:,1) = xyzr(:,:,norm_id(1):norm_id(2),3)
            end if
        end if

        ! set up temporal interpolated copy of  v, T_BA and T_AB
        allocate(v_temp(grid%n(1),grid%n(2),grid%loc_n_r,3,2))
        allocate(t_ba(grid%n(1),grid%n(2),grid%loc_n_r,9,0:0,0:0,0:0))
        allocate(t_ab(grid%n(1),grid%n(2),grid%loc_n_r,9,0:0,0:0,0:0))
        allocate(q_saf(grid%loc_n_r,0:1))
        if ((present(v_flux_tor) .or. present(v_flux_pol)) .and. &
            &.not.compare_tor_pos_loc) then
            allocate(jac(grid%n(1),grid%n(2),grid%loc_n_r))
            if (rank.eq.0 .and. .not.cont_plot) then
                if (present(v_flux_tor)) then
                    allocate(v_flux_tor(grid_trim%n(3),plot_dim(2)))
                    v_flux_tor = 0._dp
                end if
                if (present(v_flux_pol)) then
                    allocate(v_flux_pol(grid_trim%n(3),plot_dim(1)))
                    v_flux_pol = 0._dp
                end if
            end if
        end if

        call lvl_ud(-1)

        ! 1. Flux coordinates (alpha,psi,theta)
        call writo('Flux coordinates')
        call lvl_ud(1)
        if (present(v_mag)) then
            v_mag = 0._dp
            do id = 1,3
                v_mag = v_mag + v_com(:,:,:,id,1)*v_com(:,:,:,id,2)
            end do
            v_mag = sqrt(v_mag)
        end if

        ! save temporary copy, normalized
        v_temp = v_com

        ! set up plot variables
        if (do_plot .and. .not.compare_tor_pos_loc) then                        ! comparison only works for periodic quantities
            coord_names(1) = 'alpha'
            coord_names(2) = 'psi'
            if (use_pol_flux_f) then
                coord_names(3) = 'theta'
            else
                coord_names(3) = 'zeta'
            end if
            var_names = trim(base_name)
            do id = 1,3
                var_names(id,1) = trim(var_names(id,1))//'_sub_'//&
                    &trim(coord_names(id))
                var_names(id,2) = trim(var_names(id,2))//'_sup_'//&
                    &trim(coord_names(id))
            end do
            description(1) = 'covariant components of the magnetic field in &
                &Flux coordinates'
            description(2) = 'contravariant components of the magnetic field &
                &in Flux coordinates'
            description(3) = 'magnitude of the magnetic field in Flux &
                &coordinates'
            file_names(1) = trim(base_name)//'_F_sub'
            file_names(2) = trim(base_name)//'_F_sup'
            file_names(3) = trim(base_name)//'_F_mag'
            if (use_normalization) then
                v_com(:,:,:,1,1) = v_com(:,:,:,1,1) * r_0                       ! norm factor for e_alpha
                v_com(:,:,:,2,1) = v_com(:,:,:,2,1) / (r_0*b_0)                 ! norm factor for e_psi
                v_com(:,:,:,3,1) = v_com(:,:,:,3,1) * r_0                       ! norm factor for e_theta
                v_com(:,:,:,1,2) = v_com(:,:,:,1,2) / r_0                       ! norm factor for e^alpha
                v_com(:,:,:,2,2) = v_com(:,:,:,2,2) * (r_0*b_0)                 ! norm factor for e^psi
                v_com(:,:,:,3,2) = v_com(:,:,:,3,2) / r_0                       ! norm factor for e^theta
            end if
            do id = 1,2
                call plot_hdf5(var_names(:,id),trim(file_names(id)),&
                    &v_com(:,tor_id(1):tor_id(2),norm_id(1):norm_id(2),:,id),&
                    &tot_dim=plot_dim,loc_offset=plot_offset,&
                    &x=x(:,tor_id(1):tor_id(2),:,:),&
                    &y=y(:,tor_id(1):tor_id(2),:,:),&
                    &z=z(:,tor_id(1):tor_id(2),:,:),&
                    &cont_plot=cont_plot,descr=description(id))
            end do
            if (present(v_mag)) then
```

```
                call plot_hdf5(trim(base_name),trim(file_names(3)),&
                    &v_mag(:,tor_id(1):tor_id(2),norm_id(1):norm_id(2)),&
                    &tot_dim=plot_dim(1:3),loc_offset=plot_offset(1:3),&
                    &x=x(:,tor_id(1):tor_id(2),:,1),&
                    &y=y(:,tor_id(1):tor_id(2),:,1),&
                    &z=z(:,tor_id(1):tor_id(2),:,1),&
                    &cont_plot=cont_plot,descr=description(3))
            end if
        end if

        call lvl_ud(-1)

        ! 2.  Magnetic  coordinates  (phi,theta,zeta)  by  converting  using
        ! transformation matrices:
        !   (v_i)_M = T_M^F (v_i)_F
        !   (v^i)_M = (v^i)_F T_F^M
        ! where
        !             ( -q' theta 1 0 )
        !   T_MF    = (    -q     0 1 )
        !             (     1     0 0 )
        ! if the poloidal flux is used, or
        !             ( -q'/q zeta q 0 )
        !   T_MF    = (    -1      0 0 )
        !             (    1/q     0 1 )
        ! if it is the toroidal flux. Its inverse can be calculated as well.
        call writo('magnetic coordinates')
        call lvl_ud(1)
        t_ba = 0._dp
        t_ab = 0._dp
        do id = 0,1
            ierr = spline(grid_eq%loc_r_F,eq_1%q_saf_FD(:,id),grid%loc_r_F,&
                &q_saf(:,id),ord=norm_disc_prec)
            chckerr('')
        end do
        c_loc = c([1,1],.false.)
        if (use_pol_flux_f) then
            t_ba(:,:,:,c_loc,0,0,0) = -grid%theta_F
            do kd = 1,grid%loc_n_r
                t_ba(:,:,kd,c_loc,0,0,0) = t_ba(:,:,kd,c_loc,0,0,0)*q_saf(kd,1)
                t_ba(:,:,kd,c([2,1],.false.),0,0,0) = -q_saf(kd,0)
            end do
            t_ba(:,:,:,c([3,1],.false.),0,0,0) = 1._dp
            t_ba(:,:,:,c([1,2],.false.),0,0,0) = 1._dp
            t_ba(:,:,:,c([2,3],.false.),0,0,0) = 1._dp
        else
            t_ba(:,:,:,c_loc,0,0,0) = -grid%zeta_F
            do kd = 1,grid%loc_n_r
                t_ba(:,:,kd,c_loc,0,0,0) = &
                    &t_ba(:,:,kd,c_loc,0,0,0)*q_saf(kd,1)/q_saf(kd,0)
                t_ba(:,:,kd,c([2,1],.false.),0,0,0) = 1._dp/q_saf(kd,0)
                t_ba(:,:,kd,c([1,2],.false.),0,0,0) = q_saf(kd,0)
            end do
            t_ba(:,:,:,c([2,1],.false.),0,0,0) = -1._dp
            t_ba(:,:,:,c([3,3],.false.),0,0,0) = 1._dp
        end if
        ierr = calc_inv_met(t_ab,t_ba,[0,0,0])
        chckerr('')
        v_com = 0._dp
        do jd = 1,3
            do id = 1,3
                v_com(:,:,:,jd,1) = v_com(:,:,:,jd,1) + t_ba(:,:,:,&
                    &c([jd,id],.false.),0,0,0) * v_temp(:,:,:,id,1)
                v_com(:,:,:,jd,2) = v_com(:,:,:,jd,2) + t_ab(:,:,:,&
                    &c([id,jd],.false.),0,0,0) * v_temp(:,:,:,id,2)
            end do
        end do
        if (present(v_mag)) then
            v_mag = 0._dp
            do id = 1,3
                v_mag = v_mag + v_com(:,:,:,id,1)*v_com(:,:,:,id,2)
            end do
            v_mag = sqrt(v_mag)
        end if

        ! set up plot variables
        if (do_plot) then
            coord_names(1) = 'psi'
            coord_names(2) = 'theta'
            coord_names(3) = 'zeta'
            var_names = trim(base_name)
            do id = 1,3
                var_names(id,1) = trim(var_names(id,1))//'_sub_'//&
                    &trim(coord_names(id))
                var_names(id,2) = trim(var_names(id,2))//'_sup_'//&
                    &trim(coord_names(id))
            end do
            description(1) = 'covariant components of the magnetic field in &
```

```
                &Magnetic coordinates'
            description(2) = 'contravariant components of the magnetic field &
                &in Magnetic coordinates'
            description(3) = 'magnitude of the magnetic field in Magnetic &
                &coordinates'
            file_names(1) = trim(base_name)//'_M_sub'
            file_names(2) = trim(base_name)//'_M_sup'
            file_names(3) = trim(base_name)//'_M_mag'
            if (compare_tor_pos_loc) then
                do id = 1,3
                    file_names(id) = trim(file_names(id))//'_COMP'
                end do
            end if
            if (use_normalization) then
                v_com(:,:,:,1,1) = v_com(:,:,:,1,1) / (r_0*b_0)                  ! norm factor for e_psi
                v_com(:,:,:,2,1) = v_com(:,:,:,2,1) * r_0                        ! norm factor for e_theta
                v_com(:,:,:,3,1) = v_com(:,:,:,3,1) * r_0                        ! norm factor for e_zeta
                v_com(:,:,:,1,2) = v_com(:,:,:,1,2) * (r_0*b_0)                  ! norm factor for e^psi
                v_com(:,:,:,2,2) = v_com(:,:,:,2,2) / r_0                        ! norm factor for e^theta
                v_com(:,:,:,3,2) = v_com(:,:,:,3,2) / r_0                        ! norm factor for e^zeta
            end if
            if (compare_tor_pos_loc) then
                ierr = calc_tor_diff(v_com,theta_geo,norm_disc_prec)
                chckerr('')
            end if
            do id = 1,2
                call plot_hdf5(var_names(:,id),trim(file_names(id)),&
                    &v_com(:,tor_id(1):tor_id(2),norm_id(1):norm_id(2),:,id),&
                    &tot_dim=plot_dim,loc_offset=plot_offset,&
                    &x=x(:,tor_id(1):tor_id(2),:,:),&
                    &y=y(:,tor_id(1):tor_id(2),:,:),&
                    &z=z(:,tor_id(1):tor_id(2),:,:),&
                    &cont_plot=cont_plot,descr=description(id))
            end do
            if (present(v_mag)) then
                if (compare_tor_pos_loc) then
                    ierr = calc_tor_diff(v_mag,theta_geo,norm_disc_prec)
                    chckerr('')
                end if
                call plot_hdf5(trim(base_name),trim(file_names(3)),&
                    &v_mag(:,tor_id(1):tor_id(2),norm_id(1):norm_id(2)),&
                    &tot_dim=plot_dim(1:3),loc_offset=plot_offset(1:3),&
                    &x=x(:,tor_id(1):tor_id(2),:,1),&
                    &y=y(:,tor_id(1):tor_id(2),:,1),&
                    &z=z(:,tor_id(1):tor_id(2),:,1),&
                    &cont_plot=cont_plot,descr=description(3))
            end if
        end if

        if ((present(v_flux_tor) .or. present(v_flux_pol)) .and. &
            &.not.compare_tor_pos_loc) then
            ! tests
            if (maxval(grid%theta_F(grid%n(1),:,:)).lt.&
                &minval(grid%theta_F(1,:,:))) then
                call writo('theta of the grid monotomically decreases in Flux &
                    &quantities.',alert=.true.)
                call lvl_ud(1)
                call writo('This inverts the sign of the toroidal flux.')
                call writo('Remember that the grid limits are prescribed &
                    &in Flux quantities.')
                call lvl_ud(-1)
            end if
            if (maxval(grid%zeta_F(:,grid%n(2),:)).lt.&
                &minval(grid%zeta_F(:,1,:))) then
                call writo('zeta of the grid monotomically decreases in Flux &
                    &quantities.',alert=.true.)
                call lvl_ud(1)
                call writo('This inverts the sign of the poloidal flux.')
                call writo('Remember that the grid limits are prescribed &
                    &in Flux quantities.')
                if (eq_style.eq.1) call writo('For VMEC, these are inverse.')
                call lvl_ud(-1)
            end if
            if (grid_trim%r_F(grid_trim%n(3)).lt.grid_trim%r_F(1)) then
                call writo('r of the grid monotomically decreases in Flux &
                    &quantities.',alert=.true.)
                call lvl_ud(1)
                call writo('This inverts the sign of the fluxes.')
                call writo('Remember that the grid limits are prescribed &
                    &in Flux quantities.')
                call lvl_ud(-1)
            end if
            if (abs(minval(grid_trim%r_F)).gt.tol_zero) then
                call writo('r of the grid does not start at zero.',alert=.true.)
                call lvl_ud(1)
                call writo('This leaves out part of the fluxes.')
                call lvl_ud(-1)
```

```
            end if

            ! initialize temporary variable
            allocate(v_ser_temp_int(grid_trim%n(3)))

            ! set up plot variables
            var_names(1,2) = 'integrated poloidal flux of '//trim(base_name)
            var_names(2,2) = 'integrated toroidal flux of '//trim(base_name)
            file_names(1) = trim(base_name)//'_flux_pol_int'
            file_names(2) = trim(base_name)//'_flux_tor_int'

#if ldebug
            if (debug_calc_vec_comp) then
                ! user output
                call writo('Instead of calculating fluxes, returning &
                    &integrals in the angular variables.',alert=.true.)
                call lvl_ud(1)
                    call writo('Useful to check whether Maxwell holds.')
                    call writo('i.e. whether loop integral of J returns &
                        &B flux')
                    call writo('Note that the J-variables now refer to B and &
                        &vice-versa')
                    call writo('Don''t forget the contribution of the toroidal &
                        &field B_zeta on axis, times 2piR.')
                    call writo('Don''t forget the vacuum permeability!')
                call lvl_ud(-1)
            else
#endif
                ! multiply angular contravariant coordinates by Jacobian
                do jd = 1,grid_eq%n(2)
                    do id = 1,grid_eq%n(1)
                        ierr = spline(grid_eq%loc_r_F,&
                            &eq_2%jac_FD(id,jd,:,0,0,0),grid%loc_r_F,&
                            &jac(id,jd,:),ord=norm_disc_prec)
                        chckerr('')
                    end do
                end do
                if (use_normalization) jac = jac*r_0/b_0
                v_com(:,:,:,2,2) = v_com(:,:,:,2,2)*jac
                v_com(:,:,:,3,2) = v_com(:,:,:,3,2)*jac
                deallocate(jac)
#if ldebug
            end if
#endif

            ! integrate toroidal flux
            if (present(v_flux_tor) .and. grid_trim%n(1).gt.1 .and. &
                &.not.compare_tor_pos_loc) then                              ! integrate poloidally and normally
                ! loop over all toroidal points
                do jd = 1,grid_trim%n(2)
#if ldebug
                    if (debug_calc_vec_comp) then
                        ! for  all  normal   points,  integrate  poloidally  the
                        ! covariant poloidal quantity
                        do kd = norm_id(1),norm_id(2)
                            ierr = calc_int(v_com(:,jd,kd,2,1),&
                                &grid%theta_F(:,jd,kd),v_com(:,jd,kd,2,2))      ! save in contravariant variable
                            chckerr('')
                        end do

                        ! gather on master
                        ierr = get_ser_var(v_com(grid_trim%n(1),jd,&
                            &norm_id(1):norm_id(2),2,2),v_ser_temp)
                        chckerr('')

                        ! update integrals if master
                        if (rank.eq.0) then
                            if (use_pol_flux_f) then
                                v_flux_tor(:,jd) = v_flux_tor(:,jd) + v_ser_temp
                            else
                                v_flux_tor(:,jd+plot_offset(1)) = v_ser_temp
                            end if
                        end if
                    else
#endif
                        ! for all normal points, integrate poloidally
                        do kd = norm_id(1),norm_id(2)
                            ierr = calc_int(v_com(:,jd,kd,3,2),&
                                &grid%theta_F(:,jd,kd),v_com(:,jd,kd,3,1))      ! save in covariant variable
                            chckerr('')
                        end do

                        ! gather on master
                        ierr = get_ser_var(v_com(grid_trim%n(1),jd,&
                            &norm_id(1):norm_id(2),3,1),v_ser_temp)
                        chckerr('')

```

```
                        ! integrate normally and update integrals if master
                        if (rank.eq.0) then
                            ierr = calc_int(v_ser_temp,&
                                &grid_trim%r_F(norm_id(1):),v_ser_temp_int)
                            chckerr('')
                            if (use_normalization) v_ser_temp_int = &
                                &v_ser_temp_int*psi_0
                            if (use_pol_flux_f) then
                                v_flux_tor(:,jd) = v_flux_tor(:,jd) + &
                                    &v_ser_temp_int
                            else
                                v_flux_tor(:,jd+plot_offset(1)) = v_ser_temp_int
                            end if
                        end if
#if ldebug
                    end if
#endif
                end do

                ! plot output
                if (rank.eq.0 .and. do_plot .and. &
                    &eq_job_nr.eq.size(eq_jobs_lims,2)) then
                    call print_ex_2d([var_names(2,2)],&
                        &trim(file_names(2)),v_flux_tor,&
                        &x=reshape(grid_trim%r_F(norm_id(1):)*2*pi/max_flux_f,&
                        &[grid_trim%n(3),1]),draw=.false.)
                    call draw_ex([var_names(2,2)],trim(file_names(2)),&
                        &plot_dim(2),1,.false.)
                end if
            end if

            ! integrate poloidal flux
            if (present(v_flux_pol) .and. grid_trim%n(2).gt.1 .and. &
                &.not.compare_tor_pos_loc) then                              ! integrate toroidally and normally
                ! loop over all poloidal points
                do id = 1,grid_trim%n(1)
#if ldebug
                    if (debug_calc_vec_comp) then
                        ! for  all  normal   points,  integrate  toroidally  the
                        ! covariant toroidal quantity
                        do kd = norm_id(1),norm_id(2)
                            ierr = calc_int(v_com(id,:,kd,3,1),&
                                &grid%zeta_F(id,:,kd),v_com(id,:,kd,3,2))       ! save in contravariant variable
                            chckerr('')
                        end do

                        ! gather on master
                        ierr = get_ser_var(v_com(id,grid_trim%n(2),&
                            &norm_id(1):norm_id(2),3,2),v_ser_temp)
                        chckerr('')

                        ! update integrals if master
                        if (rank.eq.0) then
                            if (use_pol_flux_f) then
                                v_flux_pol(:,id+plot_offset(1)) = v_ser_temp
                            else
                                v_flux_pol(:,id) = v_flux_pol(:,id) + v_ser_temp
                            end if
                        end if
                    else
#endif
                        ! for all normal points, integrate toroidally
                        do kd = norm_id(1),norm_id(2)
                            ierr = calc_int(v_com(id,:,kd,2,2),&
                                &grid%zeta_F(id,:,kd),v_com(id,:,kd,2,1))       ! save in covariant variable
                            chckerr('')
                        end do

                        ! gather on master
                        ierr = get_ser_var(v_com(id,grid_trim%n(2),&
                            &norm_id(1):norm_id(2),2,1),v_ser_temp)
                        chckerr('')

                        ! integrate normally and update integrals if master
                        if (rank.eq.0) then
                            ierr = calc_int(v_ser_temp,&
                                &grid_trim%r_F(norm_id(1):),v_ser_temp_int)
                            chckerr('')
                            if (use_normalization) v_ser_temp_int = &
                                &v_ser_temp_int*psi_0
                            if (use_pol_flux_f) then
                                v_flux_pol(:,id+plot_offset(1)) = v_ser_temp_int
                            else
                                v_flux_pol(:,id) = v_flux_pol(:,id) + &
                                    &v_ser_temp_int
                            end if
                        end if
```

```
#if ldebug
                    end if
#endif
                end do

                ! plot output
                if (rank.eq.0 .and. do_plot .and. &
                    &eq_job_nr.eq.size(eq_jobs_lims,2)) then
                    call print_ex_2d([var_names(1,2)],&
                        &trim(file_names(1)),v_flux_pol,&
                        &x=reshape(grid_trim%r_F(norm_id(1):)*2*pi/max_flux_f,&
                        &[grid_trim%n(3),1]),draw=.false.)
                    call draw_ex([var_names(1,2)],trim(file_names(1)),&
                        &plot_dim(1),1,.false.)
                end if
            end if

            ! clean up
            deallocate(v_ser_temp)
            if (rank.eq.0) deallocate(v_ser_temp_int)
        end if

        call lvl_ud(-1)

        if (max_transf_loc.eq.2) return

        ! 3.  HELENA  coordinates  (psi,theta,zeta)  or  VMEC  coordinates
        ! (r,theta,phi), by converting using transformation matrices
        !   (v_i)_H = T_H^F (v_i)_F
        !   (v^i)_H = (v^i)_F T_F^H
        ! Note: This is  done directly from step 1; The results  from step 2 are
        ! skipped, i.e. v_temp is not overwritten after step 2.
        call writo('Equilibrium coordinates')
        call lvl_ud(1)
        do ld = 1,9
            do jd = 1,grid_eq%n(2)
                do id = 1,grid_eq%n(1)
                    ierr = spline(grid_eq%loc_r_F,&
                        &eq_2%T_FE(id,jd,:,ld,0,0,0),grid%loc_r_F,&
                        &t_ab(id,jd,:,ld,0,0,0),ord=norm_disc_prec)
                    chkerr('')
                    ierr = spline(grid_eq%loc_r_F,&
                        &eq_2%T_EF(id,jd,:,ld,0,0,0),grid%loc_r_F,&
                        &t_ba(id,jd,:,ld,0,0,0),ord=norm_disc_prec)
                    chkerr('')
                end do
            end do
        end do
        v_com = 0._dp
        do jd = 1,3
            do id = 1,3
                v_com(:,:,:,jd,1) = v_com(:,:,:,jd,1) + &
                    &t_ba(:,:,:,c([jd,id],.false.),0,0,0) * &
                    &v_temp(:,:,:,id,1)
                v_com(:,:,:,jd,2) = v_com(:,:,:,jd,2) + &
                    &t_ab(:,:,:,c([id,jd],.false.),0,0,0) * &
                    &v_temp(:,:,:,id,2)
            end do
        end do
        if (present(v_mag)) then
            v_mag = 0._dp
            do id = 1,3
                v_mag = v_mag + v_com(:,:,:,id,1)*v_com(:,:,:,id,2)
            end do
            v_mag = sqrt(v_mag)
        end if

        ! save temporary copy, normalized
        v_temp = v_com

        ! set up plot variables
        if (do_plot) then
            select case (eq_style)
                case (1)                                                ! VMEC
                    coord_names(1) = 'r'
                    coord_names(2) = 'theta'
                    coord_names(3) = 'phi'
                    description(1) = 'covariant components of the magnetic &
                        &field in VMEC coordinates'
                    description(2) = 'contravariant components of the magnetic &
                        &field in VMEC coordinates'
                    description(3) = 'magnitude of the magnetic field in VMEC &
                        &coordinates'
                    file_names(1) = trim(base_name)//'_V_sub'
                    file_names(2) = trim(base_name)//'_V_sup'
                    file_names(3) = trim(base_name)//'_V_mag'
                case (2)                                                ! HELENA
```

```
                    coord_names(1) = 'psi'
                    coord_names(2) = 'theta'
                    coord_names(3) = 'phi'
                    description(1) = 'covariant components of the magnetic &
                        &field in HELENA coordinates'
                    description(2) = 'contravariant components of the magnetic &
                        &field in HELENA coordinates'
                    description(3) = 'magnitude of the magnetic field in &
                        &HELENA coordinates'
                    file_names(1) = trim(base_name)//'_H_sub'
                    file_names(2) = trim(base_name)//'_H_sup'
                    file_names(3) = trim(base_name)//'_H_mag'
            end select
            if (compare_tor_pos_loc) then
                do id = 1,3
                    file_names(id) = trim(file_names(id))//'_COMP'
                end do
            end if
            var_names = trim(base_name)
            do id = 1,3
                var_names(id,1) = trim(var_names(id,1))//'_sub_'//&
                    &trim(coord_names(id))
                var_names(id,2) = trim(var_names(id,2))//'_sup_'//&
                    &trim(coord_names(id))
            end do
            if (use_normalization) then
                v_com(:,:,:,1,1) = v_com(:,:,:,1,1) * r_0                        ! norm factor for e_r or e_psi
                v_com(:,:,:,2,1) = v_com(:,:,:,2,1) * r_0                        ! norm factor for e_theta
                v_com(:,:,:,3,1) = v_com(:,:,:,3,1) * r_0                        ! norm factor for e_zeta or e_phi
                v_com(:,:,:,1,2) = v_com(:,:,:,1,2) / r_0                        ! norm factor for e^r or e^psi
                v_com(:,:,:,2,2) = v_com(:,:,:,2,2) / r_0                        ! norm factor for e^theta
                v_com(:,:,:,3,2) = v_com(:,:,:,3,2) / r_0                        ! norm factor for e^zeta or e^phi
            end if
            if (compare_tor_pos_loc) then
                ierr = calc_tor_diff(v_com,theta_geo,norm_disc_prec)
                chckerr('')
            end if
            do id = 1,2
                call plot_hdf5(var_names(:,id),trim(file_names(id)),&
                    &v_com(:,tor_id(1):tor_id(2),norm_id(1):norm_id(2),:,id),&
                    &tot_dim=plot_dim,loc_offset=plot_offset,&
                    &x=x(:,tor_id(1):tor_id(2),:,:),&
                    &y=y(:,tor_id(1):tor_id(2),:,:),&
                    &z=z(:,tor_id(1):tor_id(2),:,:),&
                    &cont_plot=cont_plot,descr=description(id))
            end do
            if (present(v_mag)) then
                if (compare_tor_pos_loc) then
                    ierr = calc_tor_diff(v_mag,theta_geo,norm_disc_prec)
                    chckerr('')
                end if
                call plot_hdf5(trim(base_name),trim(file_names(3)),&
                    &v_mag(:,tor_id(1):tor_id(2),norm_id(1):norm_id(2)),&
                    &tot_dim=plot_dim(1:3),loc_offset=plot_offset(1:3),&
                    &x=x(:,tor_id(1):tor_id(2),:,1),&
                    &y=y(:,tor_id(1):tor_id(2),:,1),&
                    &z=z(:,tor_id(1):tor_id(2),:,1),&
                    &cont_plot=cont_plot,descr=description(3))
            end if
        end if

        call lvl_ud(-1)

        if (max_transf_loc.eq.3) return

        ! 4.   cylindrical    coordinates   (R,phi,Z)   by    converting   using
        ! transformation matrices:
        !   (v_i)_C = T_C^E (v_i)_E
        !   (v^i)_C = (v^i)_E T_E^C
        ! where for VMEC (E=V), the transformation matrix T_VC is tabulated, and
        ! its inverse  is calculate,  and for  HELENA (E=H),  the transformation
        ! matrix T_HC is given by
        !             (    R'    0    Z'   )
        !   T_HC    = ( R_theta  0 Z_theta )
        !             (    0    -1    0    )
        ! and its inverse can be calculated as well.
        call writo('Cylindrical coordinates')
        call lvl_ud(1)
        t_ba = 0._dp
        t_ab = 0._dp
        select case (eq_style)
            case (1)                                                             ! VMEC
                do ld = 1,9
                    do jd = 1,grid_eq%n(2)
                        do id = 1,grid_eq%n(1)
                            ierr = spline(grid_eq%loc_r_F,&
                                &eq_2%T_VC(id,jd,:,ld,0,0,0),grid%loc_r_F,&
```

```
                                &t_ab(id,jd,:,ld,0,0,0),ord=norm_disc_prec)
                            chckerr('')
                        end do
                    end do
                end do
            case (2)                                                        ! HELENA
                ! setup D1R and D2R, and same thing for Z
                ! (use untrimmed grid, with ghost region)
                allocate(d1r(grid%n(1),grid%n(2),grid%loc_n_r))
                allocate(d2r(grid%n(1),grid%n(2),grid%loc_n_r))
                allocate(d1z(grid%n(1),grid%n(2),grid%loc_n_r))
                allocate(d2z(grid%n(1),grid%n(2),grid%loc_n_r))
                do jd = 1,grid%n(2)
                    do id = 1,grid%n(1)
                        ierr = spline(grid%loc_r_E,xyzr(id,jd,:,4)/norm_len,&
                            &grid%loc_r_E,d1r(id,jd,:),ord=norm_disc_prec,&
                            &deriv=1)
                        chckerr('')
                        ierr = spline(grid%loc_r_E,xyzr(id,jd,:,3)/norm_len,&
                            &grid%loc_r_E,d1z(id,jd,:),ord=norm_disc_prec,&
                            &deriv=1)
                        chckerr('')
                    end do
                end do
                do kd = 1,grid%loc_n_r
                    do jd = 1,grid%n(2)
                        ierr = spline(grid%theta_E(:,jd,kd),&
                            &xyzr(:,jd,kd,4)/norm_len,grid%theta_E(:,jd,kd),&
                            &d2r(:,jd,kd),ord=norm_disc_prec,deriv=1)
                        chckerr('')
                        ierr = spline(grid%theta_E(:,jd,kd),&
                            &xyzr(:,jd,kd,3)/norm_len,grid%theta_E(:,jd,kd),&
                            &d2z(:,jd,kd),ord=norm_disc_prec,deriv=1)
                        chckerr('')
                    end do
                end do
                t_ab(:,:,:,c([1,1],.false.),0,0,0) = d1r
                t_ab(:,:,:,c([1,3],.false.),0,0,0) = d1z
                t_ab(:,:,:,c([2,1],.false.),0,0,0) = d2r
                t_ab(:,:,:,c([2,3],.false.),0,0,0) = d2z
                t_ab(:,:,:,c([3,2],.false.),0,0,0) = -1._dp
                deallocate(d1r,d2r,d1z,d2z)
        end select
        ierr = calc_inv_met(t_ba,t_ab,[0,0,0])
        chckerr('')
        v_com = 0._dp
        do jd = 1,3
            do id = 1,3
                v_com(:,:,:,jd,1) = v_com(:,:,:,jd,1) + t_ba(:,:,:,&
                    &c([jd,id],.false.),0,0,0) * v_temp(:,:,:,id,1)
                v_com(:,:,:,jd,2) = v_com(:,:,:,jd,2) + t_ab(:,:,:,&
                    &c([id,jd],.false.),0,0,0) * v_temp(:,:,:,id,2)
            end do
        end do
        if (present(v_mag)) then
            v_mag = 0._dp
            do id = 1,3
                v_mag = v_mag + v_com(:,:,:,id,1)*v_com(:,:,:,id,2)
            end do
            v_mag = sqrt(v_mag)
        end if

        ! save temporary copy, normalized
        v_temp = v_com

        ! set up plot variables
        if (do_plot) then
            description(1) = 'covariant components of the magnetic field in &
                &cylindrical coordinates'
            description(2) = 'contravariant components of the magnetic field &
                &in cylindrical coordinates'
            description(3) = 'magnitude of the magnetic field in cylindrical &
                &coordinates'
            file_names(1) = trim(base_name)//'_C_sub'
            file_names(2) = trim(base_name)//'_C_sup'
            file_names(3) = trim(base_name)//'_C_mag'
            if (compare_tor_pos_loc) then
                do id = 1,3
                    file_names(id) = trim(file_names(id))//'_COMP'
                end do
            end if
            coord_names(1) = 'R'
            coord_names(2) = 'phi'
            coord_names(3) = 'Z'
            var_names = trim(base_name)
            do id = 1,3
                var_names(id,1) = trim(var_names(id,1))//'_sub_'//&
```

```
                &trim(coord_names(id))
            var_names(id,2) = trim(var_names(id,2))//'_sup_'//&
                &trim(coord_names(id))
        end do
        if (use_normalization) then
            !v_com(:,:,:,1,1) = v_com(:,:,:,1,1)                          ! norm factor for e_R
            v_com(:,:,:,2,1) = v_com(:,:,:,2,1) * r_0                    ! norm factor for e_phi
            !v_com(:,:,:,3,1) = v_com(:,:,:,3,1)                          ! norm factor for e_Z
            !v_com(:,:,:,1,2) = v_com(:,:,:,1,2)                          ! norm factor for e^R
            v_com(:,:,:,2,2) = v_com(:,:,:,2,2) / r_0                    ! norm factor for e^phi
            !v_com(:,:,:,3,2) = v_com(:,:,:,3,2)                          ! norm factor for e^Z
        end if
        if (compare_tor_pos_loc) then
            ierr = calc_tor_diff(v_com,theta_geo,norm_disc_prec)
            chckerr('')
        end if
        do id = 1,2
            call plot_hdf5(var_names(:,id),trim(file_names(id)),&
                &v_com(:,tor_id(1):tor_id(2),norm_id(1):norm_id(2),:,id),&
                &tot_dim=plot_dim,loc_offset=plot_offset,&
                &x=x(:,tor_id(1):tor_id(2),:,:),&
                &y=y(:,tor_id(1):tor_id(2),:,:),&
                &z=z(:,tor_id(1):tor_id(2),:,:),&
                &cont_plot=cont_plot,descr=description(id))
        end do
        if (present(v_mag)) then
            if (compare_tor_pos_loc) then
                ierr = calc_tor_diff(v_mag,theta_geo,norm_disc_prec)
                chckerr('')
            end if
            call plot_hdf5(trim(base_name),trim(file_names(3)),&
                &v_mag(:,tor_id(1):tor_id(2),norm_id(1):norm_id(2)),&
                &tot_dim=plot_dim(1:3),loc_offset=plot_offset(1:3),&
                &x=x(:,tor_id(1):tor_id(2),:,1),&
                &y=y(:,tor_id(1):tor_id(2),:,1),&
                &z=z(:,tor_id(1):tor_id(2),:,1),&
                &cont_plot=cont_plot,descr=description(3))
        end if
    end if

    call lvl_ud(-1)

    if (max_transf_loc.eq.4) return

    ! 5. Cartesian  coordinates (X,Y,Z)  by converting  using transformation
    ! matrices
    !    (e_R)    (   cos(phi)   sin(phi)  0 ) (e_X)
    !   (e_phi) = ( -R sin(phi) R cos(phi) 0 ) (e_Y)
    !    (e_Z)    (     0          0       1 ) (e_Z)
    ! with  phi  =  -  zeta_F.   For  the  transformation  of  contravariant
    ! components, the factor R becomes 1/R and the transpose is taken.
    call writo('Cartesian coordinates')
    call lvl_ud(1)
    t_ba = 0._dp
    t_ab = 0._dp
    t_ab(:,:,:,c([1,1],.false.),0,0,0) = cos(-grid%zeta_F)
    t_ab(:,:,:,c([1,2],.false.),0,0,0) = sin(-grid%zeta_F)
    t_ab(:,:,:,c([2,1],.false.),0,0,0) = -xyzr(:,:,:,4)/norm_len*&
        &sin(-grid%zeta_F)
    t_ab(:,:,:,c([2,2],.false.),0,0,0) = xyzr(:,:,:,4)/norm_len*&
        &cos(-grid%zeta_F)
    t_ab(:,:,:,c([3,3],.false.),0,0,0) = 1._dp
    ierr = calc_inv_met(t_ba,t_ab,[0,0,0])
    chckerr('')
    v_com = 0._dp
    do jd = 1,3
        do id = 1,3
            v_com(:,:,:,jd,1) = v_com(:,:,:,jd,1) + t_ba(:,:,:,&
                &c([jd,id],.false.),0,0,0) * v_temp(:,:,:,id,1)
            v_com(:,:,:,jd,2) = v_com(:,:,:,jd,2) + t_ab(:,:,:,&
                &c([id,jd],.false.),0,0,0) * v_temp(:,:,:,id,2)
        end do
    end do
    if (present(v_mag)) then
        v_mag = 0._dp
        do id = 1,3
            v_mag = v_mag + v_com(:,:,:,id,1)*v_com(:,:,:,id,2)
        end do
        v_mag = sqrt(v_mag)
    end if

    ! set up plot variables
    if (do_plot) then
        description(1) = 'covariant components of the magnetic field in &
            &cartesian coordinates'
        description(2) = 'contravariant components of the magnetic field &
            &in cartesian coordinates'
```

```
            description(3) = 'magnitude of the magnetic field in cartesian &
                &coordinates'
            file_names(1) = trim(base_name)//'_X_sub'
            file_names(2) = trim(base_name)//'_X_sup'
            file_names(3) = trim(base_name)//'_X_mag'
            if (compare_tor_pos_loc) then
                do id = 1,3
                    file_names(id) = trim(file_names(id))//'_COMP'
                end do
            end if
            coord_names(1) = 'X'
            coord_names(2) = 'Y'
            coord_names(3) = 'Z'
            var_names = trim(base_name)
            do id = 1,3
                var_names(id,1) = trim(var_names(id,1))//'_sub_'//&
                    &trim(coord_names(id))
                var_names(id,2) = trim(var_names(id,2))//'_sup_'//&
                    &trim(coord_names(id))
            end do
            if (compare_tor_pos_loc) then
                ierr = calc_tor_diff(v_com,theta_geo,norm_disc_prec)
                chckerr('')
            end if
            do id = 1,2
                call plot_hdf5(var_names(:,id),trim(file_names(id)),&
                    &v_com(:,tor_id(1):tor_id(2),norm_id(1):norm_id(2),:,id),&
                    &tot_dim=plot_dim,loc_offset=plot_offset,&
                    &x=x(:,tor_id(1):tor_id(2),:,:),&
                    &y=y(:,tor_id(1):tor_id(2),:,:),&
                    &z=z(:,tor_id(1):tor_id(2),:,:),&
                    &cont_plot=cont_plot,descr=description(id))
            end do
            if (present(v_mag)) then
                if (compare_tor_pos_loc) then
                    ierr = calc_tor_diff(v_mag,theta_geo,norm_disc_prec)
                    chckerr('')
                end if
                call plot_hdf5(trim(base_name),trim(file_names(3)),&
                    &v_mag(:,tor_id(1):tor_id(2),norm_id(1):norm_id(2)),&
                    &tot_dim=plot_dim(1:3),loc_offset=plot_offset(1:3),&
                    &x=x(:,tor_id(1):tor_id(2),:,1),&
                    &y=y(:,tor_id(1):tor_id(2),:,1),&
                    &z=z(:,tor_id(1):tor_id(2),:,1),&
                    &cont_plot=cont_plot,descr=description(3))
            end if

            ! plot vector as well
            call plot_hdf5([trim(base_name)],trim(base_name)//'_vec',&
                &v_com(:,tor_id(1):tor_id(2),norm_id(1):norm_id(2),:,1),&
                &tot_dim=plot_dim,loc_offset=plot_offset,&
                &x=x(:,tor_id(1):tor_id(2),:,:),y=y(:,tor_id(1):tor_id(2),:,:),&
                &z=z(:,tor_id(1):tor_id(2),:,:),col=4,cont_plot=cont_plot,&
                &descr='magnetic field vector')
        end if

        call lvl_ud(-1)

        ! clean up
        call grid_trim%dealloc()
    end function  calc_vec_comp

    !> Calculates the local  number of parallel grid points  for this Richardson
    !! level, taking into account that it ould be half the actual number.
    !!
    !! \see calc_ang_grid_eq_b().
    !!
    !! \return ierr
    integer function calc_n_par_x_rich(n_par_X_rich,only_half_grid) result(ierr)
        use rich_vars, only: n_par_x

        character(*), parameter :: rout_name = 'calc_n_par_X_rich'

        ! input / output
        integer, intent(inout) :: n_par_x_rich                                  !< n_par_X for this Richardson level
        logical, intent(in), optional :: only_half_grid                         !< calculate only half grid with even points

        ! local variables
        character(len=max_str_ln) :: err_msg                                    ! error message

        ! initialize ierr
        ierr = 0

        ! possibly divide n_par_X by 2
        n_par_x_rich = n_par_x
        if (present(only_half_grid)) then
            if (only_half_grid) then
```

```
                if (mod(n_par_x,2).eq.1) then
                    n_par_x_rich = (n_par_x-1)/2
                else
                    ierr = 1
                    err_msg = 'Need odd number of points'
                    chckerr(err_msg)
                end if
            end if
        end if
    end function calc_n_par_x_rich

    !> calculates the  cosine and sine mode  numbers of a function  defined on a
    !! non-regular grid.
    !!
    !! If a plot name is provided, the modes are plotted.
    !!
    !! \note The  fundamental interval is  assumed to be \f$0\ldots  2\pi\f$ but
    !! there should be no overlap between the first and last point.
    !!
    !! \return ierr
    integer function nufft(x,f,f_F,plot_name) result(ierr)
        use num_utilities, only: order_per_fun, spline

        character(*), parameter :: rout_name = 'nufft'

        ! input / output
        real(dp), intent(in) :: x(:)                                        !< coordinate values
        real(dp), intent(in) :: f(:)                                        !< function values
        real(dp), intent(inout), allocatable :: f_f(:,:)                    !< Fourier modes for \f$\cos\f$ and \f$\sin\f$
        character(len=*), intent(in), optional :: plot_name                 !< name of possible plot

        ! local variables
        integer :: m_f                                                      ! nr. of modes
        integer :: n_x                                                      ! nr. of points
        integer :: id                                                       ! counter
        logical :: print_log = .false.                                      ! print log plot as well
        real(dp), allocatable :: x_loc(:)                                   ! local x, extended
        real(dp), allocatable :: work(:)                                    ! work array
        real(dp), allocatable :: f_loc(:)                                   ! local f, extended
        real(dp), allocatable :: f_int(:)                                   ! interpolated f, later Fourier modes
        character(len=max_str_ln) :: plot_title(2)                          ! name of plot

        ! tests
        if (size(x).ne.size(f)) then
            ierr = 1
            chckerr('x and f must have same size')
        end if
        if (maxval(x)-minval(x).gt.2*pi) then
            ierr = 1
            chckerr('x is not a fundamental interval')
        end if

        ! create local x and f that go from <0 to <2pi, with a royal overlap for
        ! interpolation
        ierr = order_per_fun(x,f,x_loc,f_loc,10)
        chckerr('')

        ! set local f and interpolate
        n_x = size(x)
        allocate(f_int(n_x))
        ierr = spline(x_loc,f_loc,[((id-1._dp)/n_x*2*pi,id=1,n_x)],f_int,&
            &ord=3)
        chckerr('')

        ! clean up
        deallocate(x_loc)
        deallocate(f_loc)

        !!do id = 1,n_x
            !!f_int(id) = -4._dp+&
                !!&3*cos((id-1._dp)/n_x*2*pi*1)+&
                !!&0.5*cos((id-1._dp)/n_x*2*pi*100)+&
                !!&1.5*cos((id-1._dp)/n_x*2*pi*101)+&
                !!&4*sin((id-1._dp)/n_x*2*pi*1)+&
                !!&2*sin((id-1._dp)/n_x*2*pi*50)+&
                !!&4.5*sin((id-1._dp)/n_x*2*pi*55)
        !!end do

        ! set up variables for fft
        m_f = (n_x-1)/2                                                     ! nr. of modes
        allocate(work(3*n_x+15))                                            ! from fftpack manual

        ! calculate fft
        call dffti(n_x,work)
        call dfftf(n_x,f_int,work)
        deallocate(work)
```

```fortran

        ! rescale
        f_int(:) = f_int(:)*2/n_x
        f_int(1) = f_int(1)/2

        ! separate cos and sine
        if (allocated(f_f)) deallocate(f_f)
        allocate(f_f(m_f+1,2))
        f_f(1,1) = f_int(1)
        f_f(1,2) = 0._dp
        f_f(2:m_f+1,1) = f_int(2:2*m_f:2)
        f_f(2:m_f+1,2) = -f_int(3:2*m_f+1:2)                                    ! routine returns - sine factors

        ! output in plot if requested
        if (present(plot_name)) then
            plot_title = ['cos','sin']
            call print_ex_2d(plot_title,plot_name,f_f,draw=.false.)
            call draw_ex(plot_title,plot_name,2,1,.false.)
            if (print_log) then
                plot_title = ['cos [log]','sin [log]']
                call print_ex_2d(plot_title,trim(plot_name)//'_log',&
                    &log10(abs(f_f)),draw=.false.)
                call draw_ex(plot_title,trim(plot_name)//'_log',2,1,.false.)
            end if
        end if
    end function nufft

    !> finds smallest range that comprises a minimum and maximum value.
    subroutine find_compr_range(r_F,lim_r,lim_id)
        ! input / output
        real(dp), intent(in) :: r_f(:)                                          !< all values of coordinate
        real(dp), intent(in) :: lim_r(2)                                        !< limiting range
        integer, intent(inout) :: lim_id(2)                                     !< limiting indices

        ! local variables
        integer :: id                                                           ! counter
        integer :: n_r                                                          ! number of points in coordinate
        real(dp), parameter :: tol = 1.e-6                                      ! tolerance for grids

        ! set n_r
        n_r = size(r_f)

        lim_id = [1,n_r]                                                        ! initialize with full range
        if (r_f(1).lt.r_f(n_r)) then                                            ! ascending r_F
            do id = 1,n_r
                if (r_f(id).le.minval(lim_r)-tol) lim_id(1) = id                ! move lower limit up
                if (r_f(n_r-id+1).ge.maxval(lim_r)+tol) &
                    &lim_id(2) = n_r-id+1                                       ! move upper limit down
            end do
        else                                                                    ! descending r_F
            do id = 1,n_r
                if (r_f(id).ge.maxval(lim_r)+tol) lim_id(1) = id                ! move lower limit up
                if (r_f(n_r-id+1).le.minval(lim_r)-tol) &
                    &lim_id(2) = n_r-id+1                                       ! move upper limit down
            end do
        end if
    end subroutine find_compr_range

    !> Calculate arclength angle.
    !!
    !! This   angle   is   based   on   the   calculation   of   the   arclength
    !! from   the  start   of   the   grid,  and   then   normalizing  it   from
    !! \f$\min(\gamma)\ldots\max(\gamma)\f$, where \f$\gamma\f$  is the parallel
    !! angle.
    !!
    !! By default, the Flux variables are used,  but this can be changed with \c
    !! use_E.
    !!
    !! \note For  field-aligned grids  the projection is  taken on  the poloidal
    !! cross-section. For non-axisymmetric equilibria, this makes little sense.
    !!
    !! \return ierr
    integer function calc_arc_angle(grid,eq_1,eq_2,arc,use_E) result(ierr)
        use eq_vars, only: eq_1_type, eq_2_type, &
            &r_0
        use num_vars, only: use_normalization, use_pol_flux_f
        use num_utilities, only: c, calc_int

        character(*), parameter :: rout_name = 'calc_arc_angle'

        ! input / output
        type(grid_type), intent(in) :: grid                                     !< equilibrium grid
        type(eq_1_type), intent(in) :: eq_1                                     !< flux equilibrium variables
        type(eq_2_type), intent(in) :: eq_2                                     !< metric equilibrium variables
        real(dp), intent(inout), allocatable :: arc(:,:,:)                      !< arclength angle
        logical, intent(in), optional :: use_e                                  !< use E variables instead of F

```

```
        ! local variables
        integer :: jd, kd                                                   ! counters
        logical :: use_e_loc                                                ! local use_E

        ! initialize ierr
        ierr = 0

        ! set local use_E
        use_e_loc = .false.
        if (present(use_e)) use_e_loc = use_e

        ! calculate arclength: int dtheta |e_theta|
        allocate(arc(grid%n(1),grid%n(2),grid%loc_n_r))
        do kd = 1,grid%loc_n_r
            do jd = 1,grid%n(2)
                if (use_e_loc) then
                    ierr = calc_int(&
                        &eq_2%g_E(:,jd,kd,c([2,2],.true.),0,0,0)**(0.5),&
                        &grid%theta_E(:,jd,kd),arc(:,jd,kd))
                    chckerr('')
                    if (use_normalization) arc(:,jd,kd) = arc(:,jd,kd) * r_0    ! not really necessary as we take fractional arc length
                    arc(:,jd,kd) = grid%theta_E(1,jd,kd) + &
                        &arc(:,jd,kd) / arc(grid%n(1),jd,kd) * &
                        &(grid%theta_E(grid%n(1),jd,kd)-grid%theta_E(1,jd,kd))
                else
                    if (use_pol_flux_f) then
                        ! e_arc = e_theta - q e_alpha
                        ierr = calc_int((&
                            &eq_2%g_FD(:,jd,kd,c([3,3],.true.),0,0,0) - &
                            &eq_2%g_FD(:,jd,kd,c([1,3],.true.),0,0,0) * &
                            &2*eq_1%q_saf_FD(kd,0) + &
                            &eq_2%g_FD(:,jd,kd,c([1,1],.true.),0,0,0) * &
                            &eq_1%q_saf_FD(kd,0)**2)**(0.5),&
                            &grid%theta_F(:,jd,kd),arc(:,jd,kd))
                        chckerr('')
                    else
                        ! e_arc = -e_alpha
                        ierr = calc_int(&
                            &(-eq_2%g_FD(:,jd,kd,c([1,1],.true.),0,0,0))&
                            &**(0.5),grid%theta_F(:,jd,kd),arc(:,jd,kd))
                        chckerr('')
                    end if
                    if (use_normalization) arc(:,jd,kd) = arc(:,jd,kd) * r_0    ! not really necessary as we take fractional arc length
                    arc(:,jd,kd) = grid%theta_F(1,jd,kd) + &
                        &arc(:,jd,kd) / arc(grid%n(1),jd,kd) * &
                        &(grid%theta_F(grid%n(1),jd,kd)-grid%theta_F(1,jd,kd))
                end if
            end do
        end do
    end function calc_arc_angle
end module grid_utilities
```

D.32 /Users/toonweyens/Code/PB3D/Modules/grid_vars.f90 File Reference

Interfaces and Types

- type grid_vars::grid_type

  Type for grids.

Modules

- module grid_vars

  Variables pertaining to the different grids used.

Functions/Subroutines

- integer function grid_vars::init_grid (grid, n, i_lim, divided)

  Initializes a new grid.
- subroutine grid_vars::dealloc_grid (grid)

  Deallocates a grid.
- integer function grid_vars::copy_grid (grid_i, grid_o)

  Deep copy of a grid.

Variables

- integer, public grid_vars::n_r_in

  nr. of normal points in input grid
- integer, public grid_vars::n_r_eq

  nr. of normal points in equilibrium grid
- integer, public grid_vars::n_r_x

  nr. of normal points in perturbation grid
- integer, public grid_vars::n_r_sol

  nr. of normal points in solution grid
- integer, public grid_vars::n_alpha

  nr. of field-lines
- real(dp), public grid_vars::min_par_x

  min. of parallel coordinate [π] in field-aligned grid
- real(dp), public grid_vars::max_par_x

  max. of parallel coordinate [π] in field-aligned grid
- real(dp), public grid_vars::min_alpha

  min. of field-line label [π] in field-aligned grid
- real(dp), public grid_vars::max_alpha

  max. of field-line label [π] in field-aligned grid
- real(dp), dimension(:), allocatable, public grid_vars::alpha

  field line label alpha
- integer, public grid_vars::n_alloc_grids

  nr. of allocated grids
- integer, public grid_vars::n_alloc_discs

  nr. of allocated discretizations

D.33 grid_vars.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Variables pertaining to the different grids used.
!-----------------------------------------------------------------------------!
module grid_vars
#include <PB3D_macros.h>
    use str_utilities
    use messages
    use num_vars, only: dp, pi, max_str_ln, weight_dp

    implicit none
    private
    public n_r_in, n_r_eq, n_r_x, n_r_sol, min_par_x, max_par_x, n_alpha, &
        &min_alpha, max_alpha, alpha
#if ldebug
    public n_alloc_grids, n_alloc_discs
```

```
#endif

    ! global variables
    integer :: n_r_in                                                   !< nr. of normal points in input grid
    integer :: n_r_eq                                                   !< nr. of normal points in equilibrium grid
    integer :: n_r_x                                                    !< nr. of normal points in perturbation grid
    integer :: n_r_sol                                                  !< nr. of normal points in solution grid
    integer :: n_alpha                                                  !< nr. of field-lines
    real(dp) :: min_par_x                                               !< min. of parallel coordinate [\f$\pi\f$] in field-aligned grid
    real(dp) :: max_par_x                                               !< max. of parallel coordinate [\f$\pi\f$] in field-aligned grid
    real(dp) :: min_alpha                                               !< min. of field-line label [\f$\pi\f$] in field-aligned grid
    real(dp) :: max_alpha                                               !< max. of field-line label [\f$\pi\f$] in field-aligned grid
    real(dp), allocatable :: alpha(:)                                   !< field line label alpha
#if ldebug
    integer :: n_alloc_grids                                            !< nr. of allocated grids \ldebug
    integer :: n_alloc_discs                                            !< nr. of allocated discretizations \ldebug
#endif

    !> Type for grids
    !!
    !! The grids are saved in the following format:
    !! <tt>(angle_1,angle_2,r)</tt>, where \c angle_1 and \c angle_2 can be any
    !! angle that completely describe a flux surface.
    !!
    !! For example, they can refer to a grid of \f$\theta\f$ and \f$\zeta\f$
    !! values, but they can also refer to (Modified) flux coordinates with a
    !! parallel angle and a field line coordinate (\f$\alpha\f$).
    !!
    !! For field-aligned grids, \c angle_1 is generally chosen equal to the
    !! parallel coordinate in the Flux coordinate system, and \c angle_2 equal
    !! to the field line label (so the second dimension of the matrices is then
    !! chosen to be of size 1 for the calculations on a single field line).
    !!
    !! At the same time, the parallel coordinate, in which integrations will
    !! have to be done, ocupies the first index. This is good for numerical
    !! efficiency.
    !!
    !! For specific equilibrium grids, such as the case for HELENA equilibria,
    !! \c angle_1 corresponds to \f$\theta\f$ and \c angle_2 to the symmetry
    !! angle \f$\zeta\f$.
    !!
    !! It is important that this order of the coordinates in space is consistent
    !! among all the variables.
    type, public :: grid_type
        integer :: n(3)                                                 !< tot nr. of points
        integer :: loc_n_r                                              !< local nr. of normal points
        integer :: i_min                                                !< min. normal index of this process in full arrays
        integer :: i_max                                                !< max. normal index of this process in full arrays
        logical :: divided                                              !< whether the grid is split up among the processes
        real(dp), pointer :: r_e(:) => null()                           !< E(quilibrium) coord. values at n points
        real(dp), pointer :: r_f(:) => null()                           !< F(lux) coord. values at n points
        real(dp), pointer :: loc_r_e(:) => null()                       !< E(quilibrium) coord. values at n points
        real(dp), pointer :: loc_r_f(:) => null()                       !< F(lux) coord. values at n points
        real(dp), pointer :: theta_e(:,:,:) => null()                   !< E(quilibrium) coord. values of first angle at n points in 3-D
        real(dp), pointer :: theta_f(:,:,:) => null()                   !< F(lux) coord. values of first angle at n points in 3-D
        real(dp), pointer :: zeta_e(:,:,:) => null()                    !< E(quilibrium) coord. values of second angle at n points in 3-D
        real(dp), pointer :: zeta_f(:,:,:) => null()                    !< F(lux) coord. values of second angle at n points in 3-D
        real(dp), allocatable :: trigon_factors(:,:,:,:,:)              !< trigonometric factor cosine for the inverse fourier transf.
#if ldebug
        real(dp) :: estim_mem_usage                                     !< estimated memory usage \ldebug
#endif
    contains
        !> initialize
        procedure :: init => init_grid
        !> copy
        procedure :: copy => copy_grid
        !> deallocate
        procedure :: dealloc => dealloc_grid
    end type

contains
    !> \public Initializes a new grid.
    !!
    !! Optionally, the local limits can be provided for a divided grid.
    !!
```

```
    !! Optionally, it can be set whether the grid is divided or not. A situation
    !! where this is  useful is when only  a subset of MPI processes  is used to
    !! calculate the solution  in SLEPC. In this case, the  extra ranks all only
    !! contain the last grid point, and the last used process has as upper limit
    !! this same grid  point. This way, all the procedures  are reusable, but in
    !! this case  if only one process  is used, this procedure  becomes confused
    !! and sets  this process  to undivided. In  most cases,  this functionality
    !! probably does not need to be used.
    !!
    !! \return ierr
    integer function init_grid(grid,n,i_lim,divided) result(ierr)
#if ldebug
        use num_vars, only: print_mem_usage, rank
#endif
        character(*), parameter :: rout_name = 'init_grid'

        ! input / output
        class(grid_type), intent(inout) :: grid                                 !< grid to be initialized
        integer, intent(in) :: n(3)                                             !< tot. nr. of points (par,r,alpha)
        integer, intent(in), optional :: i_lim(2)                               !< min. and max. local normal index
        logical, intent(in), optional :: divided                                !< divided grid or not

        ! local variables
        character(len=max_str_ln) :: err_msg                                    ! error message
        logical :: divided_loc                                                  ! local divided

        ! initialize ierr
        ierr = 0

        ! tests
        if (present(i_lim)) then
            if (i_lim(2)-i_lim(1)+1.gt.n(3)) then
                ierr = 1
                err_msg = 'The local nr. of normal points cannot be higher &
                    &than the total nr. of normal points'
                chckerr(err_msg)
            end if
        end if

#if ldebug
        ! initialize memory usage
        if (print_mem_usage) grid%estim_mem_usage = 0._dp
#endif

        ! set local divided and loc_n_r
        divided_loc = .false.
        if (present(i_lim)) then                                                ! might be divided grid
            if (i_lim(2).lt.i_lim(1)) then
                ierr = 1
                write(*,*) 'i_lim = ', i_lim
                err_msg = 'faulty i_lim'
                chckerr(err_msg)
            end if
            grid%i_min = i_lim(1)
            grid%i_max = i_lim(2)
            grid%loc_n_r = i_lim(2)-i_lim(1)+1
            if (i_lim(2)-i_lim(1)+1.lt.n(3)) divided_loc = .true.               ! only divided if difference between local and total
        else                                                                    ! certainly not divided grid
            grid%i_min = 1
            grid%i_max = n(3)
            grid%loc_n_r = n(3)
        end if
        if (present(divided)) divided_loc = divided

        grid%divided = divided_loc

        ! allocate normal variables
        grid%n = n
        allocate(grid%r_E(n(3)))
        allocate(grid%r_F(n(3)))
        if (divided_loc) then
            allocate(grid%loc_r_E(grid%loc_n_r))
            allocate(grid%loc_r_F(grid%loc_n_r))
        else
            grid%loc_r_E => grid%r_E
            grid%loc_r_F => grid%r_F
        end if

#if ldebug
        ! set estimated memory usage
        if (print_mem_usage) grid%estim_mem_usage = grid%estim_mem_usage + &
            &grid%n(3)*2
#endif

        ! allocate angular variables
        if (n(1).ne.0 .and. n(2).ne.0) then
```

```
            allocate(grid%theta_E(n(1),n(2),grid%loc_n_r))
            allocate(grid%zeta_E(n(1),n(2),grid%loc_n_r))
            allocate(grid%theta_F(n(1),n(2),grid%loc_n_r))
            allocate(grid%zeta_F(n(1),n(2),grid%loc_n_r))

#if ldebug
            ! set estimated memory usage
            if (print_mem_usage) grid%estim_mem_usage = grid%estim_mem_usage + &
                &grid%loc_n_r*n(1)*n(2)*4
#endif
        end if

#if ldebug
        ! increment n_alloc_grids
        n_alloc_grids = n_alloc_grids + 1

        ! print memory usage
        if (print_mem_usage) call writo('[rank '//trim(i2str(rank))//&
            &' - Expected memory usage of grid: '//&
            &trim(r2strt(grid%estim_mem_usage*weight_dp))//' kB]',&
            &alert=.true.)
#endif
    end function init_grid

    !> \public Deallocates a grid.
    subroutine dealloc_grid(grid)
#if ldebug
        use num_vars, only: rank, print_mem_usage
#endif
        ! input / output
        class(grid_type), intent(inout) :: grid                                 !< grid to be deallocated

        ! local variables
#if ldebug
        integer :: mem_diff                                                     ! difference in memory
        real(dp) :: estim_mem_usage                                             ! estimated memory usage

        ! memory usage before deallocation
        if (print_mem_usage) then
            mem_diff = get_mem_usage()
            estim_mem_usage = grid%estim_mem_usage
        end if
#endif

        ! deallocate allocated pointers
        deallocate(grid%r_E,grid%r_F)
        if (grid%n(1).ne.0 .and. grid%n(2).ne.0) then                           ! 3D grid
            deallocate(grid%theta_E,grid%zeta_E)
            deallocate(grid%theta_F,grid%zeta_F)
        end if
        if (grid%divided) deallocate(grid%loc_r_E,grid%loc_r_F)

        ! nullify pointers
        nullify(grid%r_E,grid%r_F)
        nullify(grid%theta_E,grid%zeta_E)
        nullify(grid%theta_F,grid%zeta_F)
        nullify(grid%loc_r_E,grid%loc_r_F)

        ! deallocate allocatable variables
        call dealloc_grid_final(grid)

#if ldebug
        ! decrement n_alloc_grids
        n_alloc_grids = n_alloc_grids - 1

        ! memory usage difference after deallocation
        if (print_mem_usage) then
            mem_diff = mem_diff - get_mem_usage()
            call writo('[Rank '//trim(i2str(rank))//' - liberated '//&
                &trim(i2str(mem_diff))//'kB deallocating grid ('//&
                &trim(i2str(nint(100*mem_diff/&
                &(estim_mem_usage*weight_dp))))//&
                &'% of estimated)]',alert=.true.)
        end if
#endif
    contains
        ! Note: intent(out) automatically deallocates the variable
        !> \private
        subroutine dealloc_grid_final(grid)
            ! input / output
            type(grid_type), intent(out) :: grid                                ! grid to be deallocated
        end subroutine dealloc_grid_final
    end subroutine dealloc_grid

    !> \public Deep copy of a grid.
    !!
    !! \note Does not copy possible trigoniometric factors.
```

```
    !!
    !! \return ierr
    integer function copy_grid(grid_i,grid_o) result(ierr)
        character(*), parameter :: rout_name = 'copy_grid'

        ! input / output
        class(grid_type), intent(in) :: grid_i                                  !< grid to be copied
        type(grid_type), intent(inout) :: grid_o                                !< copied grid

        ! initialize ierr
        ierr = 0

        ierr = init_grid(grid_o,grid_i%n,i_lim=[grid_i%i_min,grid_i%i_max],&
            &divided=grid_i%divided)
        chckerr('')
        grid_o%r_E = grid_i%r_E
        grid_o%r_F = grid_i%r_F
        grid_o%loc_r_E = grid_i%loc_r_E
        grid_o%loc_r_F = grid_i%loc_r_F
        if (associated(grid_i%theta_E)) grid_o%theta_E = grid_i%theta_E
        if (associated(grid_i%theta_F)) grid_o%theta_F = grid_i%theta_F
        if (associated(grid_i%zeta_E)) grid_o%zeta_E = grid_i%zeta_E
        if (associated(grid_i%zeta_F)) grid_o%zeta_F = grid_i%zeta_F
        if (allocated(grid_i%trigon_factors)) then
            allocate(grid_o%trigon_factors(size(grid_i%trigon_factors,1),&
                &size(grid_i%trigon_factors,2),size(grid_i%trigon_factors,3),&
                &size(grid_i%trigon_factors,4),size(grid_i%trigon_factors,5)))
            grid_o%trigon_factors = grid_i%trigon_factors
        end if
    end function copy_grid
end module grid_vars
```

D.34 /Users/toonweyens/Code/PB3D/Modules/HDF5_ops.f90 File Reference

Interfaces and Types

- interface hdf5_ops::reset_hdf5_item

  Resets an HDF5 XDMF item.

Modules

- module hdf5_ops

  Operations on HDF5 and XDMF variables.

Functions/Subroutines

- integer function, public hdf5_ops::open_hdf5_file (file_info, file_name, sym_type, descr, ind_plot, cont_plot)

  Opens an HDF5 file and accompanying xmf file and returns the handles.
- integer function, public hdf5_ops::close_hdf5_file (file_info, ind_plot, cont_plot)

  Closes an HDF5 file and writes the accompanying xmf file.
- integer function, public hdf5_ops::add_hdf5_item (file_info, xdmf_item, reset, ind_plot)

  Add an XDMF item to a HDF5 file.
- integer function, public hdf5_ops::print_hdf5_3d_data_item (dataitem_id, file_info, var_name, var, dim_tot, loc_dim, loc_offset, init_val, ind_plot, cont_plot)

  Prints an HDF5 data item.
- subroutine, public hdf5_ops::merge_hdf5_3d_data_items (merged_id, dataitem_ids, var_name, dim_↩ tot, reset, ind_plot, cont_plot)

Joins dataitems in a vector.

- subroutine, public hdf5_ops::print_hdf5_att (att_id, att_dataitem, att_name, att_center, att_type, reset, ind_plot)

  Prints an HDF5 attribute.
- subroutine, public hdf5_ops::print_hdf5_top (top_id, top_type, top_n_elem, ind_plot)

  Prints an HDF5 topology.
- subroutine, public hdf5_ops::print_hdf5_geom (geom_id, geom_type, geom_dataitems, reset, ind_plot)

  Prints an HDF5 geometry.
- integer function, public hdf5_ops::print_hdf5_grid (grid_id, grid_name, grid_type, grid_time, grid_top, grid_geom, grid_atts, grid_grids, reset, ind_plot)

  Prints an HDF5 grid.
- integer function, public hdf5_ops::create_output_hdf5 (hdf5_name)

  Creates an HDF5 output file.
- integer function, public hdf5_ops::print_hdf5_arrs (vars, pb3d_name, head_name, rich_lvl, disp_info, ind_print, remove_previous_arrs)

  Prints a series of arrays, in the form of an array of pointers, to an HDF5 file.
- integer function, public hdf5_ops::read_hdf5_arr (var, pb3d_name, head_name, var_name, rich_lvl, disp_info, lim_loc)

  Reads a PB3D output file in HDF5 format.

Variables

- logical, public hdf5_ops::debug_hdf5_ops = .false.

  set to true to debug general information
- logical, public hdf5_ops::debug_print_hdf5_arrs = .false.

  set to true to debug print_HDF5_arrs

D.35 HDF5_ops.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Operations on HDF5 and XDMF variables
!!
!! \note
!! Notes about XDMF:
!!  - Collections can be spatial or temporal.  If a variable is to be evolved in
!!  time or  if its  domain is  decomposed (e.g. the  same physical  variable is
!!  defined  in  multiple HDF5  variables),  then  the  attribute name  of  this
!!  variable has to be the same for all the grids in the collection.
!!  - If no  collection is used, but  just multiple grids, the  attribute can be
!!  different but does  not have to be, as the  different grid are distinguished
!!  by their  different grid names, not  by the attribute of  the variables they
!!  contain.
!!  -  The  selection  of  hyperslabs  is  used,  as  described  here:
!!  <http://davis.lbl.gov/Manuals/HDF5-1.8.7/UG/12_Dataspaces.html>
!!  -  Chunking  is  used  for  partial  I/O,  as  described  here:
!!  <https://www.hdfgroup.org/HDF5/doc/Advanced/Chunking/> As no  chunk cache is
!!  reused, the  w0 setting  should be  1. Furthermore, the  number of  slots is
!!  chosen to be equal  to the number of chunks. And Finally,  the chunk size is
!!  chosen to be the size of the previous dimensions, if it does not exceed 4GB
!!  - Individual writes or plots are done  now using the standard I/O driver, as
!!  using  MPI does  not allow  for more  than 2GB.  This is  important for  the
!!  HELENA  version  of PB3D  where  \c  X_2  variables  get larger  than  that.
!!  <https://www.hdfgroup.org/HDF5/doc/UG/OldHtmlSource/UG_frame08TheFile.html>
!!  - Have a look in Documentation/XDMF_HDF5.pdf for an example.
!-----------------------------------------------------------------------------!
module hdf5_ops
#include <PB3D_macros.h>
    use num_vars, only: max_str_ln, dp, plot_dir, data_dir, script_dir
```

```
    use messages
    use str_utilities, only: i2str, r2str, r2strt
    use hdf5_vars
    use hdf5

    implicit none
    private
    public print_hdf5_grid, print_hdf5_geom, print_hdf5_top, &
        &print_hdf5_att, print_hdf5_3d_data_item, open_hdf5_file, &
        &close_hdf5_file, reset_hdf5_item, add_hdf5_item, create_output_hdf5, &
        &print_hdf5_arrs, read_hdf5_arr, merge_hdf5_3d_data_items
#if ldebug
    public debug_hdf5_ops, debug_print_hdf5_arrs
#endif

#if ldebug
    ! global variables
    logical :: debug_hdf5_ops = .false.                                    !< set to true to debug general information \ldebug
    logical :: debug_print_hdf5_arrs = .false.                             !< set to true to debug print_HDF5_arrs \ldebug
#endif

    ! interfaces

    !> \public Resets an HDF5 XDMF item.
    !!
    !! \note individual  version cannot make  use of array version  because then
    !! the deallocation does not work properly.
    interface reset_hdf5_item
        !> \public
        module procedure reset_hdf5_item_ind
        !> \public
        module procedure reset_hdf5_item_arr
    end interface

contains
    !> Opens an HDF5 file and accompanying xmf file and returns the handles.
    !!
    !! Optionally, a description of the file  can be provided.
    !!
    !! Also, the plot can be done for only one process, setting the variable
    !! \c ind_plot.
    !!
    !! Furthermore,  if the  plot is  a  continuation, using  \c cont_plot,  the
    !! previous plot is opened and \c sym_type is returned.
    !!
    !! \return ierr
    integer function open_hdf5_file(file_info,file_name,sym_type,descr,&
        &ind_plot,cont_plot) result(ierr)
        use num_vars, only: rank
        use mpi
        use files_utilities, only: nextunit
        use mpi_utilities, only: broadcast_var

        character(*), parameter :: rout_name = 'open_HDF5_file'

        ! input / output
        type(hdf5_file_type), intent(inout) :: file_info                   !< info about HDF5 file
        character(len=*), intent(in) :: file_name                          !< name of HDF5 file
        integer, intent(inout), optional :: sym_type                       !< symmetry type
        character(len=*), intent(in), optional  :: descr                   !< description of file
        logical, intent(in), optional :: ind_plot                          !< whether individual plot
        logical, intent(in), optional :: cont_plot                         !< continued plot

        ! local variables
        character(len=max_str_ln) :: full_file_name                        ! full file name
        character(len=max_str_ln) :: line                                  ! line in file
        character(len=max_str_ln) :: err_msg                               ! error message
        integer(HID_T) :: hdf5_i                                           ! file identifier
        integer(HID_T) :: plist_id                                         ! property list identifier
        integer :: sym_type_loc                                            ! local symmetry type
        integer :: mpi_comm_loc                                            ! MPI Communicator used
        integer :: disable_rw                                              ! MPI info to disable read and write
        integer :: i_st(2)                                                 ! indices of sym_type in read string
        integer :: id                                                      ! counter
        logical :: ind_plot_loc                                            ! local version of ind_plot
        logical :: cont_plot_loc                                           ! local version of cont_plot

        ! initialize ierr
        ierr = 0

        ! set up local ind_plot
        ind_plot_loc = .false.
        if (present(ind_plot)) ind_plot_loc = ind_plot

        ! set up local cont_plot
```

```
        cont_plot_loc = .false.
        if (present(cont_plot)) cont_plot_loc = cont_plot

        ! set up MPI Communicator
        if (ind_plot_loc) then
            mpi_comm_loc = mpi_comm_self                                    ! individual plot
        else
            mpi_comm_loc = mpi_comm_world                                   ! default world communicator
        end if

        ! set up full file name
        full_file_name = data_dir//'/'//trim(file_name)

        ! initialize FORTRAN predefined datatypes
        call h5open_f(ierr)
        chckerr('Failed to initialize HDF5')

#if lIB
        ! disable file locking
        call mpi_info_create(disable_rw,ierr)
        chckerr('Failed to create MPI info')
        call mpi_info_set(disable_rw,'romio_ds_read','disable',ierr)
        chckerr('Failed to set MPI info')
        call mpi_info_set(disable_rw,'romio_ds_write','disable',ierr)
        chckerr('Failed to set MPI info')
#else
        disable_rw = mpi_info_null
#endif

        ! setup file access property list with parallel I/O access if needed
        call h5pcreate_f(h5p_file_access_f,plist_id,ierr)
        chckerr('Failed to create property list')
        call h5pset_fapl_mpio_f(plist_id,mpi_comm_loc,disable_rw,ierr)
        chckerr('Failed to set file access property')

#if lIB
        ! free file locking info
        call mpi_info_free(disable_rw,ierr)
        chckerr('Failed to free MPI info')
#endif

        ! create or open the file collectively.
        if (cont_plot_loc) then
            call h5fopen_f(trim(full_file_name)//'.h5',h5f_acc_rdwr_f,&
                &hdf5_i,ierr,access_prp=plist_id)
            chckerr('Failed to open file')
        else
            call h5fcreate_f(trim(full_file_name)//'.h5',h5f_acc_trunc_f,&
                &hdf5_i,ierr,access_prp=plist_id)
            err_msg = 'Failed to create file. Is '//trim(data_dir)//'/ present?'
            chckerr(err_msg)
        end if

        ! close property list
        call h5pclose_f(plist_id,ierr)
        chckerr('Failed to close property list')

        ! update file info, converting integers and setting name
        file_info%HDF5_i = hdf5_i
        file_info%name = file_name

        ! XDMF  operations: only  master if  parallel  plot or  current rank  if
        ! individual plot
        if (ind_plot_loc .or. .not.ind_plot_loc.and.rank.eq.0) then
            if (cont_plot_loc) then
                ! open accompanying xmf file
                open(nextunit(file_info%XDMF_i),status='old',action='read',&
                    &file=trim(full_file_name)//'.xmf',iostat=ierr)
                chckerr('Failed to open xmf file')

                ! read symmetry type in fourth line if requested
                if (present(sym_type)) then
                    do id = 1,4
                        read(file_info%XDMF_i,'(A)',iostat=ierr) line
                        chckerr('Failed to read file')
                    end do
                    i_st(1) = index(line,'Value="')
                    if (i_st(1).eq.0) then
                        ierr = 1
                        chckerr('Can''t find symmetry type')
                    else
                        i_st(1) = i_st(1) + 7
                    end if
                    i_st(2) = index(line(i_st(1):),'"/>') + i_st(1)
                    if (i_st(2).eq.0) then
                        ierr = 1
                        chckerr('Can''t find symmetry type')
```

```
                    else
                        i_st(2) = i_st(2) - 2
                    end if
                    read(line(i_st(1):i_st(2)),*,iostat=ierr) sym_type_loc
                    chckerr('Can''t read symmetry type')
                    if (sym_type_loc.ne.sym_type) then
                        ierr = 1
                        err_msg = 'Symmetry type of continued plot does not &
                            &match previous plot'
                        chckerr(err_msg)
                    end if
                end if
            else
                ! open accompanying xmf file
                open(nextunit(file_info%XDMF_i),status='replace',&
                    &file=trim(full_file_name)//'.xmf',iostat=ierr)
                chckerr('Failed to create xmf file')

                ! write header if group master
                write(unit=file_info%XDMF_i,fmt=xmf_fmt,iostat=ierr) &
                    &'<?xml version="1.0" ?>'
                chckerr('Failed to write')
                write(unit=file_info%XDMF_i,fmt=xmf_fmt,iostat=ierr) &
                    &'<!DOCTYPE Xdmf SYSTEM "Xdmf.dtd" []>'
                chckerr('Failed to write')
                write(unit=file_info%XDMF_i,fmt=xmf_fmt,iostat=ierr) &
                    &'<Xdmf Version="2.0">'
                chckerr('Failed to write')
                if (present(sym_type)) then
                    write(unit=file_info%XDMF_i,fmt=xmf_fmt,iostat=ierr) &
                        &'<Information Name="sym_type" &
                        &Value="'//trim(i2str(sym_type))//'"/>'
                    chckerr('Failed to write')
                end if
                write(unit=file_info%XDMF_i,fmt=xmf_fmt,iostat=ierr) &
                    &'<Domain>'
                chckerr('Failed to write')
                if (present(descr)) then
                    write(unit=file_info%XDMF_i,fmt=xmf_fmt,iostat=ierr) &
                        &'<Information Name="Description">'
                    chckerr('Failed to write')
                    write(unit=file_info%XDMF_i,fmt=xmf_fmt,iostat=ierr) &
                        &trim(descr)
                    chckerr('Failed to write')
                    write(unit=file_info%XDMF_i,fmt=xmf_fmt,iostat=ierr) &
                        &'</Information>'
                    chckerr('Failed to write')
                end if
            end if
        end if

        ! pass symmetry type to other processes if continued plot
        if (cont_plot_loc .and. .not.ind_plot_loc .and. present(sym_type)) then
            ierr = broadcast_var(sym_type)
            chckerr('')
        end if
    end function open_hdf5_file

    !> Closes an HDF5 file and writes the accompanying xmf file.
    !!
    !! \return ierr
    integer function close_hdf5_file(file_info,ind_plot,cont_plot) result(ierr)
        use num_vars, only: rank

        character(*), parameter :: rout_name = 'close_HDF5_file'

        ! input / output
        type(hdf5_file_type), intent(inout) :: file_info                        !< info about HDF5 file
        logical, intent(in), optional :: ind_plot                               !< whether individual plot
        logical, intent(in), optional :: cont_plot                              !< continued plot

        ! local variables
        character(len=max_str_ln) :: err_msg                                    ! error message
        character(len=max_str_ln) :: full_file_name                             ! full file name
        integer(HID_T) :: hdf5_i                                                ! file identifier
        logical :: ind_plot_loc                                                 ! local version of ind_plot
        logical :: cont_plot_loc                                                ! local version of cont_plot

        ! initialize ierr
        ierr = 0

        ! set up local ind_plot
        ind_plot_loc = .false.
        if (present(ind_plot)) ind_plot_loc = ind_plot

        ! set up local cont_plot
        cont_plot_loc = .false.
```

```
        if (present(cont_plot)) cont_plot_loc = cont_plot

        ! set full file name and HDF5_i (converting integers)
        full_file_name = data_dir//'/'//trim(file_info%name)
        hdf5_i = file_info%HDF5_i

        ! close the HDF5 file
        call h5fclose_f(hdf5_i,ierr)
        chckerr('failed to close HDF5 file')

        ! close FORTRAN interfaces and HDF5 library.
        call h5close_f(ierr)
        err_msg = 'Failed to close FORTRAN HDF5 interface'
        chckerr(err_msg)

        ! only master if parallel plot or current rank if individual plot
        if (ind_plot_loc .or. .not.ind_plot_loc.and.rank.eq.0) then
            ! XDMF operations: only if not continued plot
            if (.not.cont_plot_loc) then
                ! close header
                write(unit=file_info%XDMF_i,fmt=xmf_fmt,iostat=ierr) &
                    &'</Domain>'
                chckerr('Failed to write')
                write(unit=file_info%XDMF_i,fmt=xmf_fmt,iostat=ierr) &
                    &'</Xdmf>'
                chckerr('Failed to write')
            end if

            ! close accompanying xmf file
            close(file_info%XDMF_i,iostat=ierr)
            chckerr('Failed to close xmf file')

            ! user output
            if (cont_plot_loc) then
                call writo('Contributed to HDF5/XMF plot in output file "'//&
                    &trim(full_file_name)//'.xmf"')
            else
                call writo('Created HDF5/XMF plot in output file "'//&
                    &trim(full_file_name)//'.xmf"')
            end if
        end if
    end function close_hdf5_file

    !> Add an XDMF item to a HDF5 file.
    !!
    !! \note  This  should only  be  used  with  grids  (or for  topologies  and
    !! geometries that are used throughout)
    !!
    !! \return ierr
    integer function add_hdf5_item(file_info,XDMF_item,reset,ind_plot) &
        &result(ierr)
        use num_vars, only: rank

        character(*), parameter :: rout_name = 'add_HDF5_item'

        ! input / output
        type(hdf5_file_type), intent(inout) :: file_info                        !< info about HDF5 file
        type(xml_str_type), intent(inout) :: xdmf_item                          !< XDMF item to add
        logical, intent(in), optional :: reset                                  !< if .true., XDMF_item is reset
        logical, intent(in), optional :: ind_plot                               !< whether individual plot

        ! local variables
        integer :: id                                                           ! counter
        integer :: item_len                                                     ! length of item
        logical :: reset_loc                                                    ! local copy of reset
        logical :: ind_plot_loc                                                 ! local version of ind_plot

        ! initialize ierr
        ierr = 0

        ! set up local ind_plot
        ind_plot_loc = .false.
        if (present(ind_plot)) ind_plot_loc = ind_plot

        ! only group master if parallel plot or current rank if individual plot
        if (ind_plot_loc .or. .not.ind_plot_loc.and.rank.eq.0) then
            ! set item_len
            item_len = size(xdmf_item%xml_str)

            ! set local reset
            if (present(reset)) then
                reset_loc = reset
            else
                reset_loc = .false.
            end if

            ! write the strings to the file
```

```
            do id = 1,item_len
                write(unit=file_info%XDMF_i,fmt=xmf_fmt,iostat=ierr) &
                    &trim(xdmf_item%xml_str(id))
                chckerr('Failed to write HDF5 item')
            end do

            ! reset if requested
            if (reset_loc) call reset_hdf5_item(xdmf_item,ind_plot_loc)
        end if
    end function add_hdf5_item

    !> Prints an HDF5 data item.
    !!
    !! If this is a  parallel data item, the group dimension  and offset have to
    !! be specified as well.
    !!
    !! \return ierr
    integer function print_hdf5_3d_data_item(dataitem_id,file_info,var_name,&
        &var,dim_tot,loc_dim,loc_offset,init_val,ind_plot,cont_plot) &
        &result(ierr)

        use num_vars, only: rank

        character(*), parameter :: rout_name = 'print_HDF5_3D_data_item'

        ! input / output
        type(xml_str_type), intent(inout) :: dataitem_id                        !< ID of data item
        type(hdf5_file_type), intent(in) :: file_info                           !< info about HDF5 file
        character(len=*), intent(in) :: var_name                                !< name of variable
        real(dp), intent(in) :: var(:,:,:)                                      !< variable to write
        integer, intent(in) :: dim_tot(3)                                       !< total dimensions of variable
        integer, intent(in), optional :: loc_dim(3)                             !< dimensions in this group
        integer, intent(in), optional :: loc_offset(3)                          !< offset in this group
        real(dp), intent(in), optional :: init_val                              !< initial fill value
        logical, intent(in), optional :: ind_plot                               !< whether individual plot
        logical, intent(in), optional :: cont_plot                              !< continued plot

        ! local variables
        integer :: id                                                           ! counter
        integer :: loc_dim_loc(3)                                               ! local copy of loc_dim
        integer :: loc_offset_loc(3)                                            ! local copy of loc_offset
        integer(HSIZE_T) :: dimsf(3)                                            ! total dataset dimensions
        integer(HSIZE_T) :: dimsm(3)                                            ! group dataset dimensions
        integer(HID_T) :: filespace                                             ! dataspace identifier in file
        integer(HID_T) :: memspace                                              ! Dataspace identifier in memory
        integer(HID_T) :: plist_id                                              ! property list identifier
        integer(HID_T) :: dset_id                                               ! dataset identifier
        integer(HSIZE_T) :: mem_offset(3)                                       ! offset in memory
        integer(HSIZE_T) :: mem_block(3)                                        ! block size in memory
        integer(HSIZE_T) :: mem_stride(3)                                       ! stride in memory
        integer(HSIZE_T) :: mem_count(3)                                        ! nr. of repetitions of block in memory
        character(len=max_str_ln) :: dim_str                                    ! string with dimensions
        character(len=max_str_ln) :: err_msg                                    ! error message
        logical :: ind_plot_loc                                                 ! local version of ind_plot
        logical :: cont_plot_loc                                                ! local version of cont_plot
        integer(HID_T) :: hdf5_kind_64                                          ! HDF5 type corresponding to dp
#if ldebug
        integer :: istat                                                        ! status
#endif

        ! initialize ierr
        ierr = 0

        ! set up local ind_plot
        ind_plot_loc = .false.
        if (present(ind_plot)) ind_plot_loc = ind_plot

        ! set up local cont_plot
        cont_plot_loc = .false.
        if (present(cont_plot)) cont_plot_loc = cont_plot

        ! set HDF5 type corresponding to dp
        hdf5_kind_64 = h5kind_to_type(dp,h5_real_kind)

        ! set the group dimensions and offset
        ierr = check_for_parallel_3d(dim_tot,loc_dim_loc,loc_offset_loc,&
            &loc_dim,loc_offset)
        chckerr('')

        ! create the data spaces for the dataset.
        dimsf = dim_tot
        dimsm = loc_dim_loc
        call h5screate_simple_f(size(dimsf),dimsf,filespace,ierr)
        chckerr('Failed to create file space')
        call h5screate_simple_f(size(dimsm),dimsm,memspace,ierr)
        chckerr('Failed to create memory space')

```

```
        ! open or create data set
        if (cont_plot_loc) then
            ! open file data set
            call h5dopen_f(file_info%HDF5_i,trim(var_name),dset_id,ierr)
            chckerr('Failed to open file data set')

            ! get file data space
            call h5dget_space_f(dset_id,filespace,ierr)
            chckerr('Failed to get file space')
        else
            ! set initial fill value if provided and create data set
            if (present(init_val)) then
                call h5pcreate_f(h5p_dataset_create_f,plist_id,ierr)
                err_msg = 'Failed to create property list'
                chckerr(err_msg)
                call h5pset_fill_value_f(plist_id,hdf5_kind_64,init_val,ierr)
                err_msg = 'Failed to set default fill value property'
                chckerr(err_msg)
                call h5dcreate_f(file_info%HDF5_i,trim(var_name),hdf5_kind_64,&
                    &filespace,dset_id,ierr,plist_id)
                chckerr('Failed to create file data set')
                call h5pclose_f(plist_id,ierr)
                chckerr('Failed to close property list')
            else
                call h5dcreate_f(file_info%HDF5_i,trim(var_name),hdf5_kind_64,&
                    &filespace,dset_id,ierr)
                chckerr('Failed to create file data set')
            end if
        end if

        ! select hyperslab in the file.
        mem_count = 1
        mem_stride = 1
        mem_block = loc_dim_loc
        mem_offset = loc_offset_loc
        call h5sselect_hyperslab_f(filespace,h5s_select_set_f,mem_offset,&
            &mem_count,ierr,mem_stride,mem_block)
        chckerr('Failed to select hyperslab')

        ! create property list for collective dataset write
        call h5pcreate_f(h5p_dataset_xfer_f,plist_id,ierr)
        chckerr('Failed to create property list')
        if (ind_plot_loc) then
            plist_id = h5p_default_f
        else
            call h5pset_dxpl_mpio_f(plist_id,h5fd_mpio_collective_f,ierr)
            chckerr('Failed to set parallel property')
        end if

        ! write the dataset collectively.
        call h5dwrite_f(dset_id,hdf5_kind_64,var,dimsf,ierr,&
            &file_space_id=filespace,mem_space_id=memspace,xfer_prp=plist_id)
        chckerr('Failed to write data set')
        call h5pclose_f(plist_id,ierr)
        chckerr('Failed to close property list')

        ! close dataspaces.
        call h5sclose_f(filespace,ierr)
        chckerr('Unable to close file space')
        call h5sclose_f(memspace,ierr)
        chckerr('Unable to close memory space')

        ! close the dataset.
        call h5dclose_f(dset_id,ierr)
        chckerr('Failed to close data set')

        ! only group master if parallel plot or current rank if individual plot,
        ! if not continued plot
        if (.not.cont_plot_loc) then
            if (ind_plot_loc .or. .not.ind_plot_loc.and.rank.eq.0) then
                ! set up dimensions string
                dim_str = ''
                do id = 1,size(dim_tot)
                    !if (dim_tot(id).gt.1) then
                        dim_str = trim(dim_str)//' '//trim(i2str(dim_tot(id)))
                    !end if
                end do
                dataitem_id%name = 'DataItem - '//trim(var_name)
                allocate(dataitem_id%xml_str(3))
                dataitem_id%xml_str(1) = '<DataItem Dimensions="'//&
                    &trim(dim_str)//'" NumberType="Float" Precision="8" &
                    &Format="HDF">'
                dataitem_id%xml_str(2) = trim(file_info%name)//'.h5:/'//&
                    &trim(var_name)
                dataitem_id%xml_str(3) = '</DataItem>'

#if ldebug
```

```
                if (debug_hdf5_ops) write(*,*,iostat=istat) &
                    &'created data item "'//trim(dataitem_id%name)//'"'
#endif
            end if
        end if
    contains
        ! Check whether the  dimensions provided are a  valid parallel indicator
        ! or not.
        !> \private
        integer function check_for_parallel_3d(dim_tot,loc_dim_out,&
            &loc_offset_out,loc_dim_in,loc_offset_in) result(ierr)
            character(*), parameter :: rout_name = 'check_for_parallel_3D'

            ! input / output
            integer, intent(in) :: dim_tot(3)                                   ! total dimension
            integer, intent(inout) :: loc_dim_out(3), loc_offset_out(3)         ! output group dimension and offset
            integer, intent(in), optional :: loc_dim_in(3), loc_offset_in(3)    ! input group dimension and offset

            ! local variables
            character(len=max_str_ln) :: err_msg                                ! error message
            integer :: id                                                       ! counter

            ! initialize ierr
            ierr = 0

            if (present(loc_dim_in)) then
                if (.not.present(loc_offset_in)) then
                    err_msg = 'Need to specify offset as well as group &
                        &dimensions'
                    ierr = 1
                    chckerr(err_msg)
                else
                    loc_dim_out = loc_dim_in
                    loc_offset_out = loc_offset_in
                end if
                do id = 1,size(dim_tot)
                    if (loc_dim_in(id).gt.dim_tot(id)) then                     ! error in parallel file
                        err_msg = 'Total dimension '//trim(i2str(id))//&
                            &' cannot be smaller than group dimension'
                        ierr = 1
                        chckerr(err_msg)
                    end if
                end do
            else
                loc_dim_out = dim_tot
                loc_offset_out = 0
            end if
        end function check_for_parallel_3d
    end function print_hdf5_3d_data_item

    !> Joins dataitems in a vector.
    subroutine merge_hdf5_3d_data_items(merged_id,dataitem_ids,var_name,&
        &dim_tot,reset,ind_plot,cont_plot)
        use num_vars, only: rank

        ! input / output
        type(xml_str_type), intent(inout) :: merged_id                          !< ID of merged data item
        type(xml_str_type), intent(inout) :: dataitem_ids(:)                    !< ID of data items
        character(len=*), intent(in) :: var_name                                !< name of variable
        integer, intent(in) :: dim_tot(3)                                       !< total dimensions of variable
        logical, intent(in), optional :: reset                                  !< whether to reset dataitems
        logical, intent(in), optional :: ind_plot                               !< whether individual plot
        logical, intent(in), optional :: cont_plot                              !< continued plot

        ! local variables
        integer :: dataitem_len                                                 ! length of data item
        integer :: id, jd                                                       ! counters
        integer :: jd_loc                                                       ! local jd
        logical :: reset_loc                                                    ! local copy of reset
        logical :: ind_plot_loc                                                 ! local version of ind_plot
        logical :: cont_plot_loc                                                ! local version of cont_plot
        character(len=max_str_ln) :: dim_str                                    ! string with dimensions
        character(len=max_str_ln) :: fun_str                                    ! string with function
#if ldebug
        integer :: istat                                                        ! status
#endif

        ! set up local ind_plot
        ind_plot_loc = .false.
        if (present(ind_plot)) ind_plot_loc = ind_plot

        ! set up local cont_plot
        cont_plot_loc = .false.
        if (present(cont_plot)) cont_plot_loc = cont_plot

        ! set local reset
        reset_loc = .false.
```

```
        if (present(reset)) reset_loc = reset

        ! only group master if parallel plot or current rank if individual plot,
        ! if not continued plot
        if (.not.cont_plot_loc) then
            if (ind_plot_loc .or. .not.ind_plot_loc.and.rank.eq.0) then
                ! set dataitem_len
                dataitem_len = 0
                do id = 1,size(dataitem_ids)
                    dataitem_len = dataitem_len + size(dataitem_ids(id)%xml_str)
                end do

                ! set XDMF attribute ID
                merged_id%name = 'DataItem - '//trim(var_name)
                allocate(merged_id%xml_str(dataitem_len+2))

                ! set up dimensions string
                dim_str = ''
                do id = 1,size(dim_tot)
                    !if (dim_tot(id).gt.1) then
                        dim_str = trim(dim_str)//' '//trim(i2str(dim_tot(id)))
                    !end if
                end do
                dim_str = trim(dim_str)//' '//trim(i2str(size(dataitem_ids)))

                ! set up function string
                fun_str = '"JOIN('
                do id = 1,size(dataitem_ids)
                    if (id.gt.1) fun_str = trim(fun_str)//','
                    fun_str = trim(fun_str)//' $'//trim(i2str(id-1))
                end do
                fun_str = trim(fun_str)//')"'

                merged_id%xml_str(1) = '<DataItem Dimensions="'//&
                    &trim(dim_str)//'" Function='//trim(fun_str)//&
                        &' ItemType="Function" >'
                jd_loc = 2
                do id = 1,size(dataitem_ids)
                    do jd = 1,size(dataitem_ids(id)%xml_str)
                        merged_id%xml_str(jd_loc) = dataitem_ids(id)%xml_str(jd)
                        jd_loc = jd_loc+1
                    end do
                end do
                merged_id%xml_str(jd_loc) = '</DataItem>'

                ! reset if requested
                if (reset_loc) call reset_hdf5_item(dataitem_ids,ind_plot_loc)

#if ldebug
                if (debug_hdf5_ops) write(*,*,iostat=istat) &
                    &'merged data items into "'//trim(merged_id%name)//'"'
#endif
            end if
        end if
    end subroutine merge_hdf5_3d_data_items

    !> Prints an HDF5 attribute.
    subroutine print_hdf5_att(att_id,att_dataitem,att_name,att_center,att_type,&
        &reset,ind_plot)

        use num_vars, only: rank

        ! input / output
        type(xml_str_type), intent(inout) :: att_id                          !< ID of attribute
        type(xml_str_type), intent(inout) :: att_dataitem                    !< dataitem of attribute
        character(len=*), intent(in) :: att_name                             !< name of attribute
        integer, intent(in) :: att_center                                    !< center of attribute
        integer, intent(in) :: att_type                                      !< type of attribute
        logical, intent(in), optional :: reset                               !< whether to reset dataitems
        logical, intent(in), optional :: ind_plot                            !< whether individual plot

        ! local variables
        integer :: dataitem_len                                              ! length of data item
        integer :: id                                                        ! counter
        logical :: reset_loc                                                 ! local copy of reset
        logical :: ind_plot_loc                                              ! local version of ind_plot
#if ldebug
        integer :: istat                                                     ! status
#endif

        ! set up local ind_plot
        ind_plot_loc = .false.
        if (present(ind_plot)) ind_plot_loc = ind_plot

        ! only group master if parallel plot or current rank if individual plot
        if (ind_plot_loc .or. .not.ind_plot_loc.and.rank.eq.0) then
            ! set dataitem_len
```

```
            dataitem_len = size(att_dataitem%xml_str)

            ! set local reset
            reset_loc = .false.
            if (present(reset)) reset_loc = reset

            ! set XDMF attribute ID
            att_id%name = 'Attribute - '//trim(att_name)
            allocate(att_id%xml_str(dataitem_len+2))
            att_id%xml_str(1) = '<Attribute Name="'//trim(att_name)//&
                &'" AttributeType="'//trim(xdmf_att_types(att_type))//&
                &'" Center="'//trim(xdmf_center_types(att_center))//'">'
            do id = 1,dataitem_len
                att_id%xml_str(id+1) = att_dataitem%xml_str(id)
            end do
            att_id%xml_str(dataitem_len+2) = '</Attribute>'

#if ldebug
            if (debug_hdf5_ops) write(*,*,iostat=istat) 'created attribute "'//&
                &trim(att_id%name)//'"'
#endif

            ! reset if requested
            if (reset_loc) call reset_hdf5_item(att_dataitem,ind_plot_loc)
        end if
    end subroutine print_hdf5_att

    !> Prints an HDF5 topology.
    !!
    !> \note Currently only structured grids supported.
    subroutine print_hdf5_top(top_id,top_type,top_n_elem,ind_plot)
        use num_vars, only: rank

        ! input / output
        type(xml_str_type), intent(inout) ::  top_id                          !< ID of topology
        integer, intent(in) :: top_type                                       !< type
        integer, intent(in) :: top_n_elem(:)                                  !< nr. of elements
        logical, intent(in), optional :: ind_plot                             !< whether individual plot

        ! local variables
        integer :: id                                                         ! counter
        integer :: n_dims                                                     ! nr. of dimensions
        character(len=max_str_ln) :: work_str                                 ! work string
        logical :: ind_plot_loc                                               ! local version of ind_plot
#if ldebug
        integer :: istat                                                      ! status
#endif

        ! set up local ind_plot
        ind_plot_loc = .false.
        if (present(ind_plot)) ind_plot_loc = ind_plot

        ! only group master if parallel plot or current rank if individual plot
        if (ind_plot_loc .or. .not.ind_plot_loc.and.rank.eq.0) then
            ! set n_dims
            n_dims = size(top_n_elem)

            ! fill work string
            work_str = ''
            do id = n_dims,1,-1
                !if (top_n_elem(id).gt.1) then
                    work_str = trim(work_str)//' '//trim(i2str(top_n_elem(id)))
                !end if
            end do

            ! set XDMF topology ID
            top_id%name = 'Topology'
            allocate(top_id%xml_str(1))
            top_id%xml_str(1) = '<Topology TopologyType="'//&
                &trim(xdmf_top_types(top_type))//'" NumberOfElements="'//&
                &trim(work_str)//'"/>'

#if ldebug
            if (debug_hdf5_ops) write(*,*,iostat=istat) 'created topology "'//&
                &trim(top_id%name)//'"'
#endif
        end if
    end subroutine print_hdf5_top

    !> Prints an HDF5 geometry.
    subroutine print_hdf5_geom(geom_id,geom_type,geom_dataitems,reset,ind_plot)
        use num_vars, only: rank

        ! input / output
        type(xml_str_type), intent(inout) ::  geom_id                         !< ID of geometry
        integer, intent(in) :: geom_type                                      !< type of geometry
        type(xml_str_type), intent(inout) :: geom_dataitems(:)                !< data items of geometry
```

```
        logical, intent(in), optional :: reset                                  !< whether to reset dataitems
        logical, intent(in), optional :: ind_plot                               !< whether individual plot

        ! local variables
        integer :: id, jd                                                       ! counters
        integer :: id_sum                                                       ! counter
        integer, allocatable :: dataitem_len(:)                                 ! length of data item
        integer :: n_dataitems                                                  ! nr. of data items
        logical :: reset_loc                                                    ! local copy of reset
        logical :: ind_plot_loc                                                 ! local version of ind_plot
#if ldebug
        integer :: istat                                                        ! status
#endif

        ! set up local ind_plot
        ind_plot_loc = .false.
        if (present(ind_plot)) ind_plot_loc = ind_plot

        ! only group master if parallel plot or current rank if individual plot
        if (ind_plot_loc .or. .not.ind_plot_loc.and.rank.eq.0) then
            ! set n_dataitems
            n_dataitems = size(geom_dataitems)

            ! set dataitem_len for every data item
            allocate(dataitem_len(n_dataitems))
            do id = 1,n_dataitems
                dataitem_len(id) = size(geom_dataitems(id)%xml_str)
            end do

            ! set local reset
            if (present(reset)) then
                reset_loc = reset
            else
                reset_loc = .false.
            end if

            ! set XDMF geometry ID
            geom_id%name = 'Geometry'
            allocate(geom_id%xml_str(sum(dataitem_len)+2))
            geom_id%xml_str(1) = '<Geometry GeometryType="'//&
                &trim(xdmf_geom_types(geom_type))//'">'
            id_sum = 2                                                          ! starting index
            do id = 1,n_dataitems
                do jd = 1,dataitem_len(id)
                    geom_id%xml_str(id_sum) = geom_dataitems(id)%xml_str(jd)
                    id_sum = id_sum + 1
                end do
            end do
            geom_id%xml_str(id_sum) = '</Geometry>'

#if ldebug
            if (debug_hdf5_ops) write(*,*,iostat=istat) 'created geometry "'//&
                &trim(geom_id%name)//'"'
#endif

            ! reset if requested
            if (reset_loc) call reset_hdf5_item(geom_dataitems,ind_plot_loc)
        end if
    end subroutine print_hdf5_geom

    !> Prints an HDF5 grid.
    !!
    !! If  this is  is a  uniform grid,  the geometry  and topology  have to  be
    !! specified, or else it  will be assumed that it is  already present in the
    !! XDMF domain, and reused.
    !!
    !! If the grid is a collection grid,  the grids in the collection have to be
    !! specified.
    !!
    !! Optionally, also a  time can be specified (for the  grids in a collection
    !! grid).
    !!
    !! \return ierr
    integer function print_hdf5_grid(grid_id,grid_name,grid_type,grid_time,&
        &grid_top,grid_geom,grid_atts,grid_grids,reset,ind_plot) result(ierr)
        use num_vars, only: rank

        character(*), parameter :: rout_name = 'print_HDF5_grid'

        ! input / output
        type(xml_str_type), intent(inout) :: grid_id                            !< ID of grid
        character(len=*), intent(in) :: grid_name                               !< name
        integer, intent(in) :: grid_type                                        !< type
        real(dp), intent(in), optional :: grid_time                             !< time of grid
        type(xml_str_type), optional :: grid_top                                !< topology
        type(xml_str_type), optional :: grid_geom                               !< geometry
        type(xml_str_type), optional :: grid_atts(:)                            !< attributes
```

```
        type(xml_str_type), optional :: grid_grids(:)                   !< grids
        logical, intent(in), optional :: reset                          !< whether top, geom and atts and grids are reset
        logical, intent(in), optional :: ind_plot                       !< whether individual plot

        ! local variables
        integer :: id, jd                                               ! counters
        integer :: id_sum                                               ! counter
        integer :: n_grids                                              ! nr. of grids
        integer :: n_atts                                               ! nr. of attributes
        integer :: time_len                                             ! length of time
        integer :: top_len                                              ! length of topology
        integer :: geom_len                                             ! length of geometry
        integer, allocatable :: atts_len(:)                             ! lengths of attributes
        integer, allocatable :: grids_len(:)                            ! lengths of grids
        logical :: reset_loc                                            ! local copy of reset
        character(len=max_str_ln) :: err_msg                            ! error message
        logical :: ind_plot_loc                                         ! local version of ind_plot
#if ldebug
        integer :: istat                                                ! status
#endif

        ! initialize ierr
        ierr = 0

        ! set up local ind_plot
        ind_plot_loc = .false.
        if (present(ind_plot)) ind_plot_loc = ind_plot

        ! only group master if parallel plot or current rank if individual plot
        if (ind_plot_loc .or. .not.ind_plot_loc.and.rank.eq.0) then
            ! test whether the correct arguments are provided
            if (grid_type.eq.1) then                                    ! uniform grid
                ! no  requirements: topology and/or  geometry can be  defined in
                ! domain for use throughout
            else if (grid_type.eq.2 .or. grid_type.eq.3) then           ! collection grid
                if (.not.present(grid_grids)) then
                    ierr = 1
                    err_msg = 'For grid collections, the grids in the &
                        &collection have to be specified'
                    chckerr(err_msg)
                end if
            else
                ierr = 1
                err_msg = 'Grid type '//trim(i2str(grid_type))//' not supported'
                chckerr(err_msg)
            end if

            ! set geom_len, atts_len and grids_len
            time_len = 0
            if (present(grid_time)) time_len = 1
            top_len = 0
            if (present(grid_top)) then
                top_len = size(grid_top%xml_str)
            else
                if (grid_type.eq.1) top_len = 1                         ! use of general domain topology
            end if
            geom_len = 0
            if (present(grid_geom)) then
                geom_len = size(grid_geom%xml_str)
            else
                if (grid_type.eq.1) geom_len = 1                        ! use of general domain geometry
            end if
            n_atts = 0
            if (present(grid_atts)) then
                n_atts = size(grid_atts)
                allocate(atts_len(n_atts))
                atts_len = 0
                do id = 1,n_atts
                    atts_len(id) = size(grid_atts(id)%xml_str)
                end do
            else
                allocate(atts_len(0))
                atts_len = 0
            end if
            n_grids = 0
            if (present(grid_grids)) then
                n_grids = size(grid_grids)
                allocate(grids_len(n_grids))
                grids_len = 0
                do id = 1,n_grids
                    grids_len(id) = size(grid_grids(id)%xml_str)
                end do
            else
                allocate(grids_len(0))
                grids_len = 0
            end if
```

```fortran

            ! set local reset
            if (present(reset)) then
                reset_loc = reset
            else
                reset_loc = .false.
            end if

            ! set XDMF grid ID
            id_sum = 2                                                  ! starting index
            if (grid_type.eq.1) then                                    ! uniform
                grid_id%name = 'Uniform Grid - '//trim(grid_name)
                allocate(&
                    &grid_id%xml_str(time_len+top_len+geom_len+sum(atts_len)+2))
                grid_id%xml_str(1) = '<Grid Name="'//trim(grid_name)//&
                    &'" GridType="Uniform">'
                if (present(grid_time)) then                            ! time
                    grid_id%xml_str(id_sum) = '<Time Value="'//&
                        &trim(r2str(grid_time))//'" />'
                    id_sum = id_sum + 1
                end if
                if (present(grid_top)) then
                    do jd = 1,top_len                                   ! topology
                        grid_id%xml_str(id_sum) = grid_top%xml_str(jd)
                        id_sum = id_sum + 1
                    end do
                else
                    grid_id%xml_str(id_sum) = '<Topology Reference="/Xdmf/&
                        &Domain/Topology[1]"/>'
                    id_sum = id_sum + 1
                end if
                if (present(grid_geom)) then
                    do jd = 1,geom_len                                  ! geometry
                        grid_id%xml_str(id_sum) = grid_geom%xml_str(jd)
                        id_sum = id_sum + 1
                    end do
                else
                    grid_id%xml_str(id_sum) = '<Geometry Reference="/Xdmf/&
                        &Domain/Geometry[1]"/>'
                    id_sum = id_sum + 1
                end if
                do id = 1,n_atts                                        ! attributes
                    do jd = 1,atts_len(id)
                        grid_id%xml_str(id_sum) = grid_atts(id)%xml_str(jd)
                        id_sum = id_sum + 1
                    end do
                end do
            else                                                        ! collection
                grid_id%name = 'Collection Grid - '//trim(grid_name)
                allocate(grid_id%xml_str(time_len+sum(grids_len)+2))
                grid_id%xml_str(1) = '<Grid Name="'//trim(grid_name)//&
                    &'" GridType="Collection" CollectionType="'//&
                    &trim(xdmf_grid_types(grid_type))//'">'
                if (present(grid_time)) then                            ! time
                    grid_id%xml_str(id_sum) = '<Time Value="'//&
                        &trim(r2str(grid_time))//'" />'
                    id_sum = id_sum + 1
                end if
                do id = 1,n_grids                                       ! grids
                    do jd = 1,grids_len(id)
                        grid_id%xml_str(id_sum) = grid_grids(id)%xml_str(jd)
                        id_sum = id_sum + 1
                    end do
                end do
            end if
            grid_id%xml_str(id_sum) = '</Grid>'

#if ldebug
            if (debug_hdf5_ops) write(*,*,iostat=istat) 'created grid "'//&
                &trim(grid_id%name)//'"'
#endif

            ! reset if requested
            if (reset_loc) then
                if (grid_type.eq.1)  then
                    if (present(grid_top)) &
                        &call reset_hdf5_item(grid_top,ind_plot_loc)
                    if (present(grid_geom)) &
                        &call reset_hdf5_item(grid_geom,ind_plot_loc)
                    if (present(grid_atts)) &
                        &call reset_hdf5_item(grid_atts,ind_plot_loc)
                else if (grid_type.eq.2 .or. grid_type.eq.3) then
                    if (present(grid_grids)) &
                        &call reset_hdf5_item(grid_grids,ind_plot_loc)
                end if
            end if
        end if
```

```
    end function print_hdf5_grid

    !> Creates an HDF5 output file.
    integer function create_output_hdf5(HDF5_name) result(ierr)
        character(*), parameter :: rout_name = 'create_output_HDF5'

        ! input / output
        character(len=*), intent(in) :: hdf5_name                                !< name of HDF5 file

        ! local variables
        character(len=max_str_ln) :: err_msg                                     ! error message
        integer(HID_T) :: hdf5_i                                                 ! file identifier

        ! initialize ierr
        ierr = 0

        call lvl_ud(1)

        ! initialize FORTRAN predefined datatypes
        call h5open_f(ierr)
        chckerr('Failed to initialize HDF5')

        ! create the file by group master
        call h5fcreate_f(trim(hdf5_name),h5f_acc_trunc_f,hdf5_i,ierr)
        err_msg = 'Failed to create file "'//trim(hdf5_name)//'"'
        chckerr(err_msg)

        ! close the HDF5 file
        call h5fclose_f(hdf5_i,ierr)
        chckerr('failed to close HDF5 file')

        ! close FORTRAN interfaces and HDF5 library.
        call h5close_f(ierr)
        err_msg = 'Failed to close FORTRAN HDF5 interface'
        chckerr(err_msg)

        call lvl_ud(-1)

        ! user output
        call writo('HDF5 output file '//trim(hdf5_name)//' created')
    end function create_output_hdf5

    !> Prints a  series of arrays, in  the form of  an array of pointers,  to an
    !! HDF5 file.
    !!
    !! Optionally, output can  be given about the variable  being written. Also,
    !! if \c  rich_lvl is  provided, <tt>_R_[rich_lvl]</tt>  is appended  to the
    !! head name if it is > 0.
    !!
    !! Also optionally, previously encountered arrays can be removed. This is to
    !! be  used with  care, as  it may  disturb the  internal workings  of PB3D.
    !! Currently (v2.15)  it is used  only for the  specific case of  jumping to
    !! solutions for \c X_grid_style 1 or 3, when writing solutions and solution
    !! grids.
    !!
    !! \note    See   <https://www.hdfgroup.org/HDF5/doc/UG/12_Dataspaces.html>,
    !! 7.4.2.3 for an explanation of the selection of the dataspaces.
    !!
    !! \return ierr
    integer function print_hdf5_arrs(vars,PB3D_name,head_name,rich_lvl,&
        &disp_info,ind_print,remove_previous_arrs) result(ierr)
        use num_vars, only: n_procs, rank
        use mpi
        use mpi_vars, only: hdf5_lock
        use mpi_utilities, only: lock_req_acc, lock_return_acc
        use hdf5_utilities, only: set_1d_vars

        character(*), parameter :: rout_name = 'print_HDF5_arrs'

        ! input / output
        type(var_1d_type), intent(in) :: vars(:)                                 !< variables to write
        character(len=*), intent(in) :: pb3d_name                                !< name of PB3D file
        character(len=*), intent(in) :: head_name                                !< head name of variables
        integer, intent(in), optional :: rich_lvl                                !< Richardson level to append to file name
        logical, intent(in), optional :: disp_info                               !< display additional information about
variable being read
        logical, intent(in), optional :: ind_print                               !< individual write (possibly partial I/O)
        logical, intent(in), optional :: remove_previous_arrs                    !< remove previous variables if present

        ! local variables
        integer :: id, jd                                                        ! counters
        integer :: mpi_comm_loc                                                  ! MPI Communicator used
        integer :: istat                                                         ! status
        integer :: n_dims                                                        ! nr. of dimensions
        integer, allocatable :: lim_tot(:,:)                                     ! total limits
        integer, allocatable :: lim_loc(:,:)                                     ! local limits
        character(len=max_str_ln) :: err_msg                                     ! error message
```

```
        character(len=max_str_ln) :: head_name_loc                          ! local head_name
        logical :: ind_print_loc                                            ! local ind_print
        logical :: disp_info_loc                                            ! local disp_info
        logical :: remove_previous_arrs_loc                                 ! local remove_previous_arrs
        logical :: group_exists                                             ! the group exists and does not have to be created
        integer(HID_T) :: a_plist_id                                        ! access property list identifier
        integer(HID_T) :: x_plist_id                                        ! transfer property list identifier
        integer(HID_T) :: chunk_c_plist_id                                  ! chunk create property list identifier
        integer(HID_T) :: hdf5_i                                            ! file identifier
        integer(HID_T) :: hdf5_kind_64                                      ! HDF5 type corresponding to dp
        integer(HID_T) :: filespace                                         ! dataspace identifier in file
        integer(HID_T) :: memspace                                          ! Dataspace identifier in memory
        integer(HID_T) :: dset_id                                           ! dataset identifier
        integer(HID_T) :: group_id                                          ! group identifier
        integer(HID_T) :: head_group_id                                     ! head group identifier
        integer(HSIZE_T) :: dimsf(1)                                        ! total dataset dimensions
        integer(HSIZE_T) :: dimsm(1)                                        ! group dataset dimensions
        integer :: disable_rw                                               ! MPI info to disable read and write
#if ldebug
        real :: rdcc_w0                                                     ! Preemption policy.
        integer(SIZE_T) :: rdcc_nslots                                      ! Number of chunk slots in the raw data chunk  cache hash table.
        integer(SIZE_T) :: rdcc_nbytes                                      ! Total size of the raw data chunk cache, in bytes.
#endif

        ! initialize ierr
        ierr = 0

        ! set local disp_info
        disp_info_loc = .false.
        if (present(disp_info)) disp_info_loc = disp_info

        ! set up local head_name
        head_name_loc = head_name
        if (present(rich_lvl)) then
            if (rich_lvl.gt.0) head_name_loc = trim(head_name_loc)//'_R_'//&
                &trim(i2str(rich_lvl))
        end if

        ! set local remove_previous_arrs
        remove_previous_arrs_loc = .false.
        if (present(remove_previous_arrs)) remove_previous_arrs_loc = &
            &remove_previous_arrs

        ! detect whether individual or collective print
        call detect_ind_print(ind_print_loc)
        if (present(ind_print)) ind_print_loc = ind_print

        ! set up MPI Communicator
        if (ind_print_loc) then
            mpi_comm_loc = mpi_comm_self                                    ! individual print
        else
            mpi_comm_loc = mpi_comm_world                                   ! default world communicator
        end if

        ! initialize FORTRAN predefined datatypes
        call h5open_f(ierr)
        chckerr('Failed to initialize HDF5')

#if lIB
        ! disable file locking
        call mpi_info_create(disable_rw,ierr)
        chckerr('Failed to create MPI info')
        call mpi_info_set(disable_rw,'romio_ds_read','disable',ierr)
        chckerr('Failed to set MPI info')
        call mpi_info_set(disable_rw,'romio_ds_write','disable',ierr)
        chckerr('Failed to set MPI info')
#else
        disable_rw = mpi_info_null
#endif

        ! setup file access property list with parallel I/O access if needed
        call h5pcreate_f(h5p_file_access_f,a_plist_id,ierr)
        chckerr('Failed to create property list')
        if (ind_print_loc) then
            call h5pset_fapl_stdio_f(a_plist_id,ierr)
        else
            call h5pset_fapl_mpio_f(a_plist_id,mpi_comm_loc,disable_rw,ierr)
        end if
        chckerr('Failed to set file access property')

#if lIB
        ! free file locking info
        call mpi_info_free(disable_rw,ierr)
        chckerr('Failed to free MPI info')
```

```
#endif

        ! wait for file access if individual print and multiple processes
        if (n_procs.gt.1 .and. ind_print_loc) then
            ierr = lock_req_acc(hdf5_lock)
            chckerr('')
        end if

        ! open the file
        call h5fopen_f(trim(pb3d_name),h5f_acc_rdwr_f,hdf5_i,ierr,&
            &access_prp=a_plist_id)
        chckerr('Failed to open file')
        call h5pclose_f(a_plist_id,ierr)
        chckerr('Failed to close property list')

        ! set HDF5 type corresponding to dp
        hdf5_kind_64 = h5kind_to_type(dp,h5_real_kind)

        ! check if head group exists and if not create it
        call h5eset_auto_f(0,ierr)
        chckerr('Failed to disable error printing')
        call h5gopen_f(hdf5_i,trim(head_name_loc),head_group_id,istat)
        group_exists = istat.eq.0
        call h5eset_auto_f(1,ierr)
        if (group_exists .and. remove_previous_arrs_loc) then
            call h5gclose_f(head_group_id, ierr)
            chckerr('Failed to close head group')
            call h5ldelete_f(hdf5_i,trim(head_name_loc),ierr)
            chckerr('Failed to delete group')
            group_exists = .false.
#if ldebug
            if (debug_print_hdf5_arrs) call writo('group "'//&
                &trim(head_name_loc)//'" existed and was deleted')
#endif
        end if
        if (.not.group_exists) then                                     ! group does not exist
            call h5gcreate_f(hdf5_i,trim(head_name_loc),head_group_id,ierr)
            chckerr('Failed to create group')
        end if
        chckerr('Failed to enable error printing')

        ! user output
        call writo('Write data to PB3D output "'//trim(pb3d_name)//'/'//&
            &trim(head_name_loc)//'"')
        call lvl_ud(1)

        ! loop over all elements in vars
        do id = 1,size(vars)
            ! user output
            if (disp_info_loc) then
                call writo('Writing '//trim(vars(id)%var_name))
            end if

            ! check if group exists and if not create it
            call h5eset_auto_f(0,ierr)
            chckerr('Failed to disable error printing')
            call h5gopen_f(head_group_id,trim(vars(id)%var_name),group_id,istat)
            group_exists = istat.eq.0
            call h5eset_auto_f(1,ierr)
            chckerr('Failed to enable error printing')
            if (.not.group_exists) then                                 ! group does not exist
                ! create group
                call h5gcreate_f(head_group_id,trim(vars(id)%var_name),&
                        &group_id,ierr)
                chckerr('Failed to create group')
            end if

            ! 1. Data itself

            ! set up local and total limits
            n_dims = size(vars(id)%tot_i_min)
            allocate(lim_tot(n_dims,2))
            allocate(lim_loc(n_dims,2))
            lim_tot = &
                &reshape([vars(id)%tot_i_min,vars(id)%tot_i_max],[n_dims,2])
            lim_loc = &
                &reshape([vars(id)%loc_i_min,vars(id)%loc_i_max],[n_dims,2])

            ! set dimsf
            dimsf = product(lim_tot(:,2)-lim_tot(:,1)+1)                ! file space has total dimensions

            ! create or open data set with filespace
            if (.not.group_exists) then                                 ! group did not exist
                ! create file data space
                call h5screate_simple_f(1,dimsf,filespace,ierr)
                chckerr('Failed to create file space')

```

```
                ! set up chunk creation property list
                ierr = set_1d_vars(lim_tot,lim_loc,c_plist_id=chunk_c_plist_id)
                chckerr('')

                ! create file data set in group
                call h5dcreate_f(group_id,'var',hdf5_kind_64,filespace,&
                        &dset_id,ierr,dcpl_id=chunk_c_plist_id)
                chckerr('Failed to create file data set')

                ! close chunk property list
                call h5pclose_f(chunk_c_plist_id,ierr)
                chckerr('Failed to close property list')
            else                                                        ! group already exists
                ! open file data set
                call h5dopen_f(group_id,'var',dset_id,ierr)
                chckerr('Failed to open file data set')

                ! get file data space
                call h5dget_space_f(dset_id,filespace,ierr)
                chckerr('Failed to get file space')
            end if

#if ldebug
            if (debug_print_hdf5_arrs) then
                ! get data access property list
                call h5dget_access_plist_f(dset_id,a_plist_id,ierr)
                chckerr('Failed to get property list')

                ! get chunk cache
                call h5pget_chunk_cache_f(a_plist_id,rdcc_nslots,&
                    &rdcc_nbytes,rdcc_w0,ierr)
                write(*,*,iostat=istat) rank, 'Number of chunk slots in the &
                    &raw data chunk cache hash table:', rdcc_nslots
                write(*,*,iostat=istat) rank, 'Total size of the raw data &
                    &chunk cache, in Mbytes:', rdcc_nbytes*1.e-6_dp
                write(*,*,iostat=istat) rank, 'Preemption Policy:', rdcc_w0
                chckerr('Failed to get chunk cache')

                ! close data access property list
                call h5pclose_f(a_plist_id,ierr)
                chckerr('Failed to close property list')
            end if
#endif

            ! set 1D filespace hyperslab selection
            ierr = set_1d_vars(lim_tot,lim_loc,space_id=filespace)
            chckerr('')

            ! create property list for collective dataset write
            call h5pcreate_f(h5p_dataset_xfer_f,x_plist_id,ierr)
            chckerr('Failed to create property list')
            if (ind_print_loc) then
                x_plist_id = h5p_default_f
            else
                call h5pset_dxpl_mpio_f(x_plist_id,h5fd_mpio_collective_f,ierr)
                chckerr('Failed to set parallel property')
            end if

            ! set dimsm
            dimsm = product(lim_loc(:,2)-lim_loc(:,1)+1)                ! memory space has local dimensions

            ! create memory data space
            call h5screate_simple_f(1,dimsm,memspace,ierr)
            chckerr('Failed to create memory space')

            ! write the dataset collectively.
            call h5dwrite_f(dset_id,hdf5_kind_64,vars(id)%p,dimsf,ierr,&
                &file_space_id=filespace,mem_space_id=memspace,&
                &xfer_prp=x_plist_id)
            if (ierr.ne.0) then
                call writo('Did you increase max_tot_mem while restarting &
                    &Richardson or jumping to solution with different grid &
                    &size?',alert=.true.)
                call lvl_ud(1)
                call writo('If so, must restart from lvl = 1.')
                call writo('Or modify the code to use "remove_previous_arrs".')
                call lvl_ud(-1)
            end if
            chckerr('Failed to write data data set')
            call h5pclose_f(x_plist_id,ierr)
            chckerr('Failed to close property list')

            ! close dataspaces
            call h5sclose_f(filespace,ierr)
            chckerr('Unable to close file space')
            call h5sclose_f(memspace,ierr)
            chckerr('Unable to close memory space')
```

```

            ! close the dataset.
            call h5dclose_f(dset_id,ierr)
            chckerr('Failed to close data set')

            ! 2. Limit information

            ! set dimsf
            dimsf = size(lim_tot)                                           ! min. and max. limit value per dimension

            ! create or open data set with filespace
            if (.not.group_exists) then                                     ! group did not exist
                ! create file data space
                call h5screate_simple_f(1,dimsf,filespace,ierr)
                chckerr('Failed to create file space')

                ! create file data set in group
                call h5dcreate_f(group_id,'lim',h5t_native_integer,&
                    &filespace,dset_id,ierr)
                chckerr('Failed to create file data set')

                ! close dataspace
                call h5sclose_f(filespace,ierr)
                chckerr('Unable to close file space')
            else                                                            ! group already exists
                ! open file data set
                call h5dopen_f(group_id,'lim',dset_id,ierr)
                chckerr('Failed to open file data set')
            end if

            ! only leader writes
            if (rank.eq.0 .or. ind_print_loc) then
                ! write the dataset
                call h5dwrite_f(dset_id,h5t_native_integer,&
                    &[vars(id)%tot_i_min,vars(id)%tot_i_max],dimsf,ierr)

                chckerr('Failed to write limit data set')
            end if

            ! close the dataset.
            call h5dclose_f(dset_id,ierr)
            chckerr('Failed to close data set')

            ! deallocate limits
            deallocate(lim_tot,lim_loc)

            ! close the group
            call h5gclose_f(group_id,ierr)
            chckerr('Failed to close group')
        end do

        call lvl_ud(-1)

        ! close the head group
        call h5gclose_f(head_group_id,ierr)
        chckerr('Failed to close group')

        ! close the HDF5 and return lock
        call h5fclose_f(hdf5_i,ierr)
        chckerr('failed to close HDF5 file')
        if (n_procs.gt.1 .and. ind_print_loc) then
            ! return lock
            ierr = lock_return_acc(hdf5_lock)
            chckerr('')
        end if

        ! close FORTRAN interfaces and HDF5 library.
        call h5close_f(ierr)
        err_msg = 'Failed to close FORTRAN HDF5 interface'
        chckerr(err_msg)
    contains
        ! Detects whether individual print.
        !> \private
        subroutine detect_ind_print(ind_print)
            ! input / output
            logical, intent(inout) :: ind_print                             ! whether an individual or collective print

            ! initialize ind_print to true
            ind_print = .true.

            ! one process ensures individual print
            if (n_procs.eq.1) return

            ! loop over all elements in vars
            do id = 1,size(vars)
                do jd = 1,size(vars(id)%tot_i_min)
                    if (vars(id)%tot_i_min(jd).ne.vars(id)%loc_i_min(jd) .or. &
```

```
                        &vars(id)%tot_i_max(jd).ne.vars(id)%loc_i_max(jd)) then
                        ind_print = .false.
                        return
                    end if
                end do
            end do
        end subroutine
    end function print_hdf5_arrs

    !> Reads a PB3D output file in HDF5 format.
    !!
    !! This happens in  a non-parallel way. By default, all  variables are read,
    !! but an array of strings with acceptable variable names can be passed.
    !!
    !! Optionally, output can  be given about the variable being  read. Also, if
    !! \c rich_lvl  is provided,  <tt>_R_rich_lvl</tt> is  appended to  the head
    !! name if it is > 0.
    !!
    !! Furthermore, using \c lim_loc, a hyperslab of the variable can be read.
    !!
    !! \note If a limit of lim_loc is a negative value, the procedure just takes
    !! the entire range.  This is necessary as sometimes  the calling procuderes
    !! don't have,  and don't need to  have, knowledge of the  underlying sizes,
    !! for example in the case of having multiple parallel jobs.
    !!
    !! \return ierr
    integer function read_hdf5_arr(var,PB3D_name,head_name,var_name,rich_lvl,&
        &disp_info,lim_loc) result(ierr)
        use hdf5_utilities, only: set_1d_vars
        use mpi_vars, only: hdf5_lock
        use mpi_utilities, only: lock_req_acc, lock_return_acc
#if ldebug
        use hdf5_utilities, only: list_all_vars_in_group
#endif

        character(*), parameter :: rout_name = 'read_HDF5_arr_ind'

        ! input / output
        type(var_1d_type), intent(inout) :: var                                 !< variable to read
        character(len=*), intent(in) :: pb3d_name                               !< name of PB3D file
        character(len=*), intent(in) :: head_name                               !< head name of variables
        character(len=*), intent(in) :: var_name                                !< name of variable
        integer, intent(in), optional :: rich_lvl                               !< Richardson level to reconstruct
        logical, intent(in), optional :: disp_info                              !< display additional information about variable being read
        integer, intent(in), optional :: lim_loc(:,:)                           !< local limits

        ! local variables
        character(len=max_str_ln) :: err_msg                                    ! error message
        integer(HID_T) :: hdf5_i                                                ! file identifier
        integer(HID_T) :: dset_id                                               ! dataset identifier
        integer(HID_T) :: hdf5_kind_64                                          ! HDF5 type corresponding to dp
        integer(HID_T) :: group_id                                              ! group identifier
        integer(HID_T) :: head_group_id                                         ! head group identifier
        integer(HID_T) :: filespace                                             ! dataspace identifier in file
        integer(HID_T) :: memspace                                              ! Dataspace identifier in memory
        integer(HSIZE_T) :: id                                                  ! counter
        integer(HSIZE_T) :: data_size                                           ! size of data set
        integer(HSIZE_T) :: n_dims                                              ! nr. of dimensions
        integer(SIZE_T) :: name_len                                             ! length of name of group
        integer :: storage_type                                                 ! type of storage used in HDF5 file
        integer :: nr_lnks_head                                                 ! number of links in head group
        integer :: max_corder                                                   ! current maximum creation order value for group
        integer, allocatable :: lim_tot(:,:)                                    ! total limits
        integer, allocatable :: lim_loc_loc(:,:)                                ! local lim_loc
        character(len=max_str_ln) :: name_len_loc                               ! local copy of name_len
        character(len=max_str_ln) :: group_name                                 ! name of group
        character(len=max_str_ln) :: head_name_loc                              ! local head_name
        logical :: disp_info_loc                                                ! local disp_info
#if ldebug
        integer :: istat                                                        ! status
#endif

        ! initialize ierr
        ierr = 0

        ! set local disp_info
        disp_info_loc = .false.
        if (present(disp_info)) disp_info_loc = disp_info

        ! set up local head_name
        head_name_loc = head_name
        if (present(rich_lvl)) then
            if (rich_lvl.gt.0) head_name_loc = trim(head_name_loc)//'_R_'//&
                &trim(i2str(rich_lvl))
        end if
```

```fortran

        ! user output
#if ldebug
        if (debug_hdf5_ops) then
            write(*,*,iostat=istat) 'Reading data from PB3D output "'//&
                &trim(pb3d_name)//'/'//trim(head_name_loc)//'/'//&
                &trim(var_name)//'"'
        end if
#endif

        ! initialize FORTRAN predefined datatypes
        call h5open_f(ierr)
        chckerr('Failed to initialize HDF5')

        ! preparation
        hdf5_kind_64 = h5kind_to_type(dp,h5_real_kind)

        ! wait for file access in a non-blocking way
        ierr = lock_req_acc(hdf5_lock,blocking=.false.)
        chckerr('')

        ! user output
        if (disp_info_loc) then
            call writo('Opening file '//trim(pb3d_name))
        end if

        ! open the file
        call h5fopen_f(trim(pb3d_name),h5f_acc_rdonly_f,hdf5_i,ierr)
        chckerr('Failed to open file')

        ! user output
        if (disp_info_loc) then
            call writo('Opening variable "'//trim(head_name_loc)//'"')
        end if

        ! open head group
        call h5gopen_f(hdf5_i,trim(head_name_loc),head_group_id,ierr)
        err_msg = 'Failed to open head group "'//trim(head_name_loc)//'"'
        chckerr(err_msg)

#if ldebug
        ! display all variables in group
        if (disp_info_loc) then
            ierr = list_all_vars_in_group(head_group_id)
            chckerr('')
        end if
#endif

        ! get number of objects in group to allocate vars
        call h5gget_info_f(head_group_id,storage_type,nr_lnks_head,&
            &max_corder,ierr)
        chckerr('Failed to get group info')

        ! iterate over all elements in group to save all acceptable variables
        do id = 1, nr_lnks_head
            call h5lget_name_by_idx_f(head_group_id,'.',h5_index_name_f,&
                &h5_iter_native_f,id-1,group_name,ierr,size=name_len)
            chckerr('Failed to get name')

#if ldebug
            ! check length
            if (name_len.gt.max_str_ln) then
                write(name_len_loc,*) name_len
                ierr = 1
                err_msg = 'Recompile with max_str_ln > '//trim(name_len_loc)
                chckerr(err_msg)
            end if
#endif

            if (trim(group_name).ne.trim(var_name)) then                        ! not found
                cycle
            end if

            ! user output
            if (disp_info_loc) then
                call writo('Reading '//trim(group_name))
            end if

            ! set name
            var%var_name = trim(group_name)

            ! open group
            call h5gopen_f(head_group_id,trim(group_name),group_id,ierr)

            chckerr('Failed to open group')

            ! 1. limit information
```

```

            ! open limit information dataset
            call h5dopen_f(group_id,'lim',dset_id,ierr)
            chckerr('Failed to open dataset')

            ! get dataspace
            call h5dget_space_f(dset_id,filespace,ierr)
            chckerr('Failed to get file space')

            ! get size to allocate the 1D variable
            call h5sget_simple_extent_npoints_f(filespace,data_size,ierr)

            chckerr('Failed to get storage size')
            n_dims = data_size/2
            allocate(var%tot_i_min(n_dims))
            allocate(var%tot_i_max(n_dims))

            ! close dataspace
            call h5sclose_f(filespace,ierr)
            chckerr('Failed to close file space')

            ! allocate helper variables
            allocate(lim_tot(n_dims,2))
            allocate(lim_loc_loc(n_dims,2))

            ! read into 1D variable
            call h5dread_f(dset_id,h5t_native_integer,lim_tot,[2*n_dims],&
                &ierr)
            chckerr('Failed to read dataset')

            ! set up local limits
            if (present(lim_loc)) then
                lim_loc_loc = lim_loc
                where (lim_loc(:,1).lt.0) lim_loc_loc(:,1) = lim_tot(:,1)
                where (lim_loc(:,2).lt.0) lim_loc_loc(:,2) = lim_tot(:,2)
            else
                lim_loc_loc = lim_tot
            end if

            ! copy into limits of var
            ! Note: tot_i_min is started from 1,  the actual limits in the grid,
            ! eq_1, eq_2,  X_1, X_2 or  sol variables  is set by  the respective
            ! "init" procedure.
            var%tot_i_min = 1
            var%tot_i_max = lim_loc_loc(:,2)-lim_loc_loc(:,1)+1

            ! close the dataset.
            call h5dclose_f(dset_id,ierr)
            chckerr('Failed to close data set')

            ! 2. variable

            ! set up local data size to allocate the 1D variable
            data_size = product(lim_loc_loc(:,2)-lim_loc_loc(:,1)+1)
            allocate(var%p(data_size))

            ! open variable dataset
            call h5dopen_f(group_id,'var',dset_id,ierr)
            chckerr('Failed to open dataset')

            ! get dataspace
            call h5dget_space_f(dset_id,filespace,ierr)
            chckerr('Failed to get file space')

            ! set 1D filespace hyperslab selection
            ierr = set_1d_vars(lim_tot,lim_loc_loc,space_id=filespace)
            chckerr('')

            ! create memory data space
            call h5screate_simple_f(1,[data_size],memspace,ierr)
            chckerr('Failed to create memory space')

            ! read into 1D variable
            call h5dread_f(dset_id,hdf5_kind_64,var%p,[data_size],&
                &ierr,mem_space_id=memspace,file_space_id=filespace)
            chckerr('Failed to read dataset')

            ! close dataspaces
            call h5sclose_f(filespace,ierr)
            chckerr('Failed to close file space')
            call h5sclose_f(memspace,ierr)
            chckerr('Unable to close memory space')

            ! close the dataset.
            call h5dclose_f(dset_id,ierr)
            chckerr('Failed to close data set')

```

```
            ! end of group

            ! deallocate limits
            deallocate(lim_tot,lim_loc_loc)

            ! close the group
            call h5gclose_f(group_id,ierr)
            chckerr('Failed to close group')

            ! exit loop
            exit
        end do

        ! close the head group
        call h5gclose_f(head_group_id,ierr)
        chckerr('Failed to close head group')

        ! close the HDF5 file
        call h5fclose_f(hdf5_i,ierr)
        chckerr('failed to close HDF5 file')

        ! return lock
        ierr = lock_return_acc(hdf5_lock)
        chckerr('')

        ! close FORTRAN interfaces and HDF5 library.
        call h5close_f(ierr)
        err_msg = 'Failed to close FORTRAN HDF5 interface'
        chckerr(err_msg)
    end function read_hdf5_arr

    !> \private array version
    subroutine reset_hdf5_item_arr(XDMF_items,ind_plot)
        use num_vars, only: rank

        ! input / output
        type(xml_str_type), intent(inout) :: XDMF_items(:)                      !< XDMF items to reset
        logical, intent(in), optional :: ind_plot                               !< whether individual plot

        ! local variables
        integer :: id                                                           ! counter
        integer :: n_items                                                      ! nr. of items
        logical :: ind_plot_loc                                                 ! local version of ind_plot
#if ldebug
        integer :: istat                                                        ! status
#endif

        ! set up local ind_plot
        ind_plot_loc = .false.
        if (present(ind_plot)) ind_plot_loc = ind_plot

        ! set n_items
        n_items = size(xdmf_items)

        ! only group master if parallel plot or current rank if individual plot
        if (ind_plot_loc .or. .not.ind_plot_loc.and.rank.eq.0) then
            do id = 1,n_items
                if (.not.allocated(xdmf_items(id)%xml_str)) then
                    call writo('Could not reset HDF5 XDMF item "'&
                        &//trim(xdmf_items(id)%name)//'"',warning=.true.)
                else
#if ldebug
                    if (debug_hdf5_ops) write(*,*,iostat=istat) 'reset "'//&
                        &trim(xdmf_items(id)%name)//'"'
#endif
                    xdmf_items(id)%name = ''
                    deallocate(xdmf_items(id)%xml_str)
                end if
            end do
        end if
    end subroutine reset_hdf5_item_arr
    !> \private individual version
    subroutine reset_hdf5_item_ind(XDMF_item,ind_plot)
        use num_vars, only: rank

        ! input / output
        type(xml_str_type), intent(inout) :: XDMF_item                          !< XDMF item to reset
        logical, intent(in), optional :: ind_plot                               !< whether individual plot

        ! local variables
        logical :: ind_plot_loc                                                 ! local version of ind_plot
#if ldebug
        integer :: istat                                                        ! status
#endif

        ! set up local ind_plot
        ind_plot_loc = .false.
```

```
        if (present(ind_plot)) ind_plot_loc = ind_plot

        ! only group master if parallel plot or current rank if individual plot
        if (ind_plot_loc .or. .not.ind_plot_loc.and.rank.eq.0) then
            if (.not.allocated(xdmf_item%xml_str)) then
                call writo('Could not reset HDF5 XDMF item "'&
                    &//trim(xdmf_item%name)//'"',warning=.true.)
            else
#if ldebug
                if (debug_hdf5_ops) write(*,*,iostat=istat) 'reset "'//&
                    &trim(xdmf_item%name)//'"'
#endif
                xdmf_item%name = ''
                deallocate(xdmf_item%xml_str)
            end if
        end if
    end subroutine reset_hdf5_item_ind
end module hdf5_ops
```

D.36 /Users/toonweyens/Code/PB3D/Modules/HDF5_utilities.f90 File Reference

Modules

- module hdf5_utilities

  Utilities pertaining to HDF5 and XDMF.

Functions/Subroutines

- integer function, public hdf5_utilities::set_1d_vars (lim_tot, lim_loc, space_id, c_plist_id)

  Sets up the chunk property list and/or the 1D hyperslabs that correspond to a local subset of `lim_tot` given by the limits `lim_loc`.
- integer function, public hdf5_utilities::probe_hdf5_group (hdf5_name, group_name, group_exists)

  Probe HDF5 file for group existence.
- integer function, public hdf5_utilities::list_all_vars_in_group (group_id)

  Lists all variables in a HDF5 group.

Variables

- logical, public hdf5_utilities::debug_set_1d_vars = .false.

  set to true to debug set_1D_vars

D.37 HDF5_utilities.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Utilities pertaining to HDF5 and XDMF
!!
!! \see See hdf5_ops for notes on HDF5 and XDMF.
!-----------------------------------------------------------------------------!
module hdf5_utilities
#include <PB3D_macros.h>
    use num_vars, only: max_str_ln, dp, plot_dir, data_dir, script_dir
    use messages
    use str_utilities, only: i2str, r2str, r2strt
    use hdf5_vars
```

```
    use hdf5

    implicit none
    private
    public probe_hdf5_group, set_1d_vars
#if ldebug
    public list_all_vars_in_group, &
        &debug_set_1d_vars
#endif

#if ldebug
    ! global variables
    logical :: debug_set_1d_vars = .false.                                  !< set to true to debug set_1D_vars \ldebug
#endif

contains
    !> Sets up the chunk property list  and/or the 1D hyperslabs that correspond
    !! to a local subset of \c lim_tot given by the limits \c lim_loc.
    !!
    !! An example  for the  hyperslab is  given in  3 dimensions  with size  n =
    !! 5,3,3:
    !!  - for dim = 1, limits: 2..4
    !!  - for dim = 2, limits: 3..3
    !!  - for dim = 3, limits: 2..3.
    !!
    !! The 1D equivalent can be represented as
    !! □□□
    !! [.....][.....][.....] | [.....][.....][.....] | [.....][.....][.....]
    !! □□□
    !! Now, dimension 1 will only allow for the following elements
    !! (given by <tt>-</tt>):
    !! □□□
    !! [.---.][.---.][.---.] | [.---.][.---.][.---.] | [.---.][.---.][.---.]
    !! □□□
    !! dimension 2 will only allow for the following elements:
    !! □□□
    !! [.....][.....][-----] | [.....][.....][-----] | [.....][.....][-----]
    !! □□□
    !! dimension 3 will only allow for the following elements:
    !! □□□
    !! [.....][.....][.....] | [-----][-----][-----] | [-----][-----][-----]
    !! □□□
    !! so that the total is given by:
    !! □□□
    !! [.....][.....][.....] | [.....][.....][.....] | [.....][.....][.---.]
    !! □□□
    !!
    !! In practice, it is easiest to work with a full selection, where for every
    !! dimension  the  ranges that  do  not  correspond  to  the range  of  that
    !! dimension are  taken away. Clearly,  if a  dimension has the  full range,
    !! nothing will be taken out:
    !!  - \c block_i     = \f$n_1 n_2 \cdots n_{i-1} (b_i-a_{i+1}) \f$ ,
    !!  - \c stride_i    = \f$n_1 n_2 \cdots n_{i-1} (B_i-A_{i+1}) \f$ ,
    !!  - \c offset_i    = \f$n_1 n_2 \cdots n_{i-1} (a_i-A_i) \f$ ,
    !!  - \c count_i     = \f$n_{i+1} n_{i+2} \cdots n_N \f$ ,
    !!
    !! where \f$a_i\f$  and \f$b_i\f$  represent the  local limits  of dimension
    !! \f$i\f$,  \f$A_i\f$  and \f$B_i\f$  the  total  ones  and the  number  of
    !! dimensions is \f$N\f$, as can be verified.
    !!
    !! The chunk property  list for creation can  be set up so that  its size is
    !! equal to the largest dimensions of full range, or an integer part of that
    !! if it exceeds  4GB. Since the variables  don't need to be  used more than
    !! once,  either \c  nbytes or  \c nslots  could be  set to  0 in  an access
    !! property list, but this is currently not done.
    !!
    !! \return ierr
    integer function set_1d_vars(lim_tot,lim_loc,space_id,c_plist_id) &
        &result(ierr)
#if ldebug
        use num_vars, only: rank
#endif

        character(*), parameter :: rout_name = 'set_1D_vars'

        ! input / output
        integer, intent(in) :: lim_tot(:,:)                                 !< total limits
        integer, intent(in) :: lim_loc(:,:)                                 !< local limits
        integer(HID_T), intent(in), optional :: space_id                    !< dataspace identifier
        integer(HID_T), intent(inout), optional :: c_plist_id               !< chunk creation property list identifier

        ! local variables
        integer :: id, kd                                                   ! counters
        integer :: n_dims                                                   ! nr. of dimensions
        integer :: n_prod(2)                                                ! n_prod(1) = n_1*n_2*..*n_(i-1), n_prod(2)
    = n_prod(1)*n_i
        integer :: div_dim                                                  ! first divided dimension
```

```
        integer :: chunk_size                                           ! size of chunk
        integer(HSIZE_T) :: block(1)                                    ! block size in memory
        integer(HSIZE_T) :: stride(1)                                   ! stride in memory
        integer(HSIZE_T) :: count(1)                                    ! nr. of repetitions of block in memory
        integer(HSIZE_T) :: offset(1)                                   ! offset in memory
        integer(HSIZE_T) :: chunk_dims(1)                               ! chunk dimensions
        real(dp) :: max_chunk_size                                      ! maximum 4 GB (from manual)
        real(dp) :: min_chunk_size                                      ! minimum 10 MB (from experience)
        character(len=max_str_ln) :: err_msg                            ! error message
#if ldebug
        integer :: istat                                                ! status
#endif

        ! initialize ierr
        ierr = 0

        ! set n_dims
        n_dims = size(lim_tot,1)

#if ldebug
        if (size(lim_loc,1).ne.n_dims) then
            ierr = 1
            err_msg = 'lim_tot and lim_loc are not compatible'
            chckerr(err_msg)
        end if
        if (size(lim_tot,2).ne.2 .or. size(lim_loc,2).ne.2) then
            ierr = 1
            err_msg = 'lim_tot and_lim_loc need to contain 2 columns'
            chckerr(err_msg)
        end if
        do id = 1,n_dims
            if (lim_tot(id,1).gt.lim_loc(id,1) .or. &
                &lim_tot(id,2).lt.lim_loc(id,2)) then
                write(*,*) rank, 'lim_tot = ', lim_tot
                write(*,*) rank, 'lim_loc = ', lim_loc
                ierr = 1
                err_msg = 'Total limits must comprise local ones'
                chckerr(err_msg)
            end if
        end do

        if (debug_set_1d_vars) then
            write(*,*,iostat=istat) rank, 'in set_1D_vars:'
            write(*,*,iostat=istat) rank, 'lim_tot_lo = ', lim_tot(:,1)
            write(*,*,iostat=istat) rank, 'lim_tot_hi = ', lim_tot(:,2)
            write(*,*,iostat=istat) rank, 'lim_loc_lo = ', lim_loc(:,1)
            write(*,*,iostat=istat) rank, 'lim_loc_hi = ', lim_loc(:,2)
        end if
#endif

        ! initialize divided dimension
        div_dim = n_dims+1

        ! set hyperslab to be everything if requested
        if (present(space_id)) then
            call h5sselect_all_f(space_id,ierr)
            chckerr('Failed to select hyperslab')
        end if

        ! loop over dimensions to possibly restrict hyperslab and set div_dim
        do id = 1,n_dims
            ! only restrict hyperslab if local range differ from total one
            if (lim_loc(id,1).gt.lim_tot(id,1) .or. &
                &lim_loc(id,2).lt.lim_tot(id,2)) then
                ! set divided dimension if first time
                if (div_dim.eq.n_dims+1) div_dim = id

                ! calculations if hyperslab restriction requested
                if (present(space_id)) then
                    ! set auxiliary variables
                    n_prod(1) = product(lim_tot(1:id-1,2)-lim_tot(1:id-1,1)+1)
                    n_prod(2) = n_prod(1)*(lim_tot(id,2)-lim_tot(id,1)+1)

#if ldebug
                    if (debug_set_1d_vars) then
                        write(*,*,iostat=istat) rank, 'dimension', id, 'of', &
                            &n_dims
                        write(*,*,iostat=istat) rank, 'n_prod = ', n_prod
                    end if
#endif

                    block = (lim_loc(id,2)-lim_loc(id,1)+1) * n_prod(1)
                    stride = n_prod(2)
                    offset = (lim_loc(id,1)-lim_tot(id,1)) * n_prod(1)
                    count = 1
                    do kd = id+1,n_dims
                        count = count*(lim_tot(kd,2)-lim_tot(kd,1)+1)
```

```
                    end do

#if ldebug
                    if (debug_set_1d_vars) then
                        write(*,*,iostat=istat) rank, 'block, offset, stride, &
                            &count = ', block, offset, stride, count
                    end if
#endif

                    call h5sselect_hyperslab_f(space_id,h5s_select_and_f,&
                        &offset,count,ierr,stride=stride,block=block)
                    chckerr('Failed to select hyperslab')
                end if
#if ldebug
                if (debug_set_1d_vars) then
                    write(*,*,iostat=istat) rank, 'total range DIFFERS from &
                        &local'
                end if
            else
                if (debug_set_1d_vars) then
                    write(*,*,iostat=istat) rank, 'total range EQUAL to local'
                end if
#endif
            end if
        end do

        ! set chunk property list if requested
        max_chunk_size = 4.e9_dp / sizeof(1._dp)                                ! maximum 4 GB (from manual)
        min_chunk_size = min(10.e6_dp/sizeof(1._dp),&
            &product(lim_tot(:,2)-lim_tot(:,1)+1._dp))                          ! maximum 10 MB (from experience), or limit to total variable size
        if (present(c_plist_id)) then
            ! set up variables
            chunk_size = 1
            id = 1
            do while (chunk_size.le.min_chunk_size .and. id.le.size(lim_tot,1))
                chunk_size = chunk_size * (lim_tot(id,2)-lim_tot(id,1)+1)
                id = id + 1
            end do
#if ldebug
            if (debug_set_1d_vars) then
                write(*,*,iostat=istat) rank, 'suggested chunk size', chunk_size
            end if
#endif
            chunk_size = chunk_size / (1 + floor(chunk_size/max_chunk_size))    ! limit to max chunk size
#if ldebug
            if (debug_set_1d_vars) then
                write(*,*,iostat=istat) rank, 'def. chunk size', chunk_size
            end if
#endif

            ! set creation property list if provided
            if (present(c_plist_id)) then
                chunk_dims = chunk_size
                call h5pcreate_f(h5p_dataset_create_f,c_plist_id,ierr)
                chckerr('Failed to create property list')
                call h5pset_chunk_f(c_plist_id,1,chunk_dims,ierr)               ! one dimension for 1D chunk
                chckerr('Failed to create chunk')
            end if
        end if
    end function set_1d_vars

    !> Probe HDF5 file for group existence.
    !!
    !! \return ierr
    integer function probe_hdf5_group(HDF5_name,group_name,group_exists) &
        &result(ierr)
        use mpi_vars, only: hdf5_lock
        use mpi_utilities, only: lock_req_acc, lock_return_acc

        character(*), parameter :: rout_name = 'probe_HDF5_group'

        ! input / output
        character(len=*), intent(in) :: hdf5_name                               !< name of HDF5 file
        character(len=*), intent(in) :: group_name                              !< name of group to probe for
        logical, intent(inout) :: group_exists                                  !< whether group exists

        ! local variables
        integer :: istat                                                        ! status
        integer(HID_T) :: hdf5_i                                                ! file identifier
        integer(HID_T) :: group_id                                              ! group identifier

        ! initialize ierr
        ierr = 0

        ! initialize FORTRAN predefined datatypes
        call h5open_f(ierr)
```

```
        chckerr('Failed to initialize HDF5')

        ! wait for file access in a non-blocking way
        ierr = lock_req_acc(hdf5_lock,blocking=.false.)
        chckerr('')

        ! open the file
        call h5fopen_f(hdf5_name,h5f_acc_rdonly_f,hdf5_i,ierr)
        chckerr('Failed to open file. Does it exist?')

        ! disable error messages
        call h5eset_auto_f(0,ierr)
        chckerr('Failed to disable error printing')

        ! try to open group
        call h5gopen_f(hdf5_i,group_name,group_id,istat)
        group_exists = istat.eq.0

        ! reenable error messages
        call h5eset_auto_f(1,ierr)
        chckerr('Failed to enable error printing')

        ! close group
        if (group_exists) then
            call h5gclose_f(group_id,ierr)
            chckerr('Failed to close head group')
        end if

        ! close the HDF5 file
        call h5fclose_f(hdf5_i,ierr)
        chckerr('failed to close HDF5 file')

        ! return lock
        ierr = lock_return_acc(hdf5_lock)
        chckerr('')

        ! close FORTRAN interfaces and HDF5 library.
        call h5close_f(ierr)
        chckerr('Failed to close FORTRAN HDF5 interface')
    end function probe_hdf5_group

#if ldebug
    !> Lists all variables in a HDF5 group.
    !!
    !! \ldebug
    !!
    !! \return ierr
    integer function list_all_vars_in_group(group_id) result(ierr)
        character(*), parameter :: rout_name = 'list_all_vars_in_group'

        ! input / output
        integer(HID_T), intent(in) :: group_id                                  !< group identifier

        ! local variables
        integer :: storage_type                                                 ! type of storage used in HDF5 file
        integer :: nr_lnks                                                      ! number of links in group
        integer :: max_corder                                                   ! current maximum creation order value for group
        integer(HSIZE_T) :: id                                                  ! counter
        integer :: id_loc                                                       ! local id
        integer(SIZE_T) :: name_len                                             ! length of name of group
        character(len=max_str_ln) :: var_name                                   ! name of variable

        ! get number of objects in group
        call h5gget_info_f(group_id,storage_type,nr_lnks,max_corder,&
            &ierr)
        chckerr('Failed to get group info')

        ! user output
        call writo('The group has '//trim(i2str(nr_lnks))//' elements')

        ! loop over all elements
        do id = 1,nr_lnks
            ! get variable name
            call h5lget_name_by_idx_f(group_id,'.',h5_index_name_f,&
                &h5_iter_native_f,id-1,var_name,ierr,size=name_len)
            chckerr('Failed to get name')

            ! user output
            id_loc = int(id,1)
            call writo('Element '//trim(i2str(id_loc))//'/'//&
                &trim(i2str(nr_lnks))//': '//trim(var_name))
        end do
    end function list_all_vars_in_group
#endif
end module hdf5_utilities
```

D.38 /Users/toonweyens/Code/PB3D/Modules/HDF5_vars.f90 File Reference

Interfaces and Types

- type hdf5_vars::xml_str_type

  XML strings used in XDMF.
- type hdf5_vars::hdf5_file_type

  HDF5 data type, containing the information about HDF5 files.
- type hdf5_vars::var_1d_type

  1D equivalent of multidimensional variables, used for internal HDF5 storage.
- interface hdf5_vars::dealloc_xml_str

  Deallocates XML_str_type.
- interface hdf5_vars::dealloc_var_1d

  Deallocates 1D variable.

Modules

- module hdf5_vars

  Variables pertaining to HDF5 and XDMF.

Functions/Subroutines

- subroutine, public hdf5_vars::init_hdf5

  Initializes the HDF5 types.

Variables

- character(len=6), public hdf5_vars::xmf_fmt = '(999A)'

  format to write the xmf file
- integer, parameter, public hdf5_vars::max_dim_var_1d = 100000

  maximum dimension of var_1D
- character(len=max_str_ln), dimension(2), public hdf5_vars::xdmf_num_types

  possible XDMF number types
- character(len=max_str_ln), dimension(2), public hdf5_vars::xdmf_format_types

  possible XDMF format types
- character(len=max_str_ln), dimension(2), public hdf5_vars::xdmf_geom_types

  possible XDMF geometry types
- character(len=max_str_ln), dimension(2), public hdf5_vars::xdmf_top_types

  possible XDMF topology types
- character(len=max_str_ln), dimension(2), public hdf5_vars::xdmf_att_types

  possible XDMF attribute types
- character(len=max_str_ln), dimension(2), public hdf5_vars::xdmf_center_types

  possible XDMF attribute center types
- character(len=max_str_ln), dimension(3), public hdf5_vars::xdmf_grid_types

  possible XDMF grid types

D.39 HDF5_vars.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Variables pertaining to HDF5 and XDMF.
!-----------------------------------------------------------------------------!
module hdf5_vars
#include <PB3D_macros.h>
    use num_vars, only: max_str_ln, dp, plot_dir, data_dir, script_dir
    use messages
    use str_utilities, only: i2str, r2str, r2strt
    use hdf5

    implicit none
    private
    public init_hdf5, dealloc_xml_str, dealloc_var_1d, &
        &xmf_fmt, xdmf_num_types, xdmf_format_types, xdmf_geom_types, &
        &xdmf_top_types, xdmf_att_types, xdmf_center_types, xdmf_grid_types, &
        &max_dim_var_1d

    ! global variables
    integer, parameter :: max_xml_ln = 300                                  !< max. length of xml string
    character(len=6) :: xmf_fmt = '(999A)'                                  !< format to write the xmf file
    integer, parameter :: max_dim_var_1d = 100000                           !< maximum dimension of var_1D

    ! XDMF possibilities
    character(len=max_str_ln) :: xdmf_num_types(2)                          !< possible XDMF number types
    character(len=max_str_ln) :: xdmf_format_types(2)                       !< possible XDMF format types
    character(len=max_str_ln) :: xdmf_geom_types(2)                         !< possible XDMF geometry types
    character(len=max_str_ln) :: xdmf_top_types(2)                          !< possible XDMF topology types
    character(len=max_str_ln) :: xdmf_att_types(2)                          !< possible XDMF attribute types
    character(len=max_str_ln) :: xdmf_center_types(2)                       !< possible XDMF attribute center types
    character(len=max_str_ln) :: xdmf_grid_types(3)                         !< possible XDMF grid types

    !> XML strings used in XDMF
    type, public :: xml_str_type
        character(len=max_str_ln) :: name                                   !< name of this item
        integer :: max_xml_ln = 300                                         !< max. length of xml string
        character(len=max_xml_ln), allocatable :: xml_str(:)                !< XML string
    end type xml_str_type

    !> HDF5 data type, containing the information about HDF5 files.
    type, public :: hdf5_file_type
        integer(HID_T) :: hdf5_i                                            !< HDF5 file handle
        integer :: xdmf_i                                                   !< XDMF file handle
        character(len=max_str_ln) :: name                                   !< name of files (without extensions ".h5" and ".xmf")
    end type hdf5_file_type

    !> 1D equivalent of multidimensional variables, used for internal HDF5
    !! storage.
    type, public :: var_1d_type
        real(dp), allocatable :: p(:)                                       !< 1D equivalent of data of variable
        integer, allocatable :: tot_i_min(:)                                !< total min.of indices of variable
        integer, allocatable :: tot_i_max(:)                                !< total max.of indices of variable
        integer, allocatable :: loc_i_min(:)                                !< group min.of indices of variable
        integer, allocatable :: loc_i_max(:)                                !< group max.of indices of variable
        character(len=max_str_ln) :: var_name                               !< name of variable
    end type var_1d_type

    ! interfaces

    !> \public Deallocates XML_str_type.
    interface dealloc_xml_str
        !> \public
        module procedure dealloc_xml_str_ind
        !> \public
        module procedure dealloc_xml_str_arr
    end interface

    !> \public Deallocates 1D variable.
    interface dealloc_var_1d
        !> \public
        module procedure dealloc_var_1d_ind
        !> \public
        module procedure dealloc_var_1d_arr
        !> \public
        module procedure dealloc_var_1d_arr_2
    end interface

contains
    !> Initializes the HDF5 types.
    !!
    !! Has to be called once.
```

```
    subroutine init_hdf5
        ! XDMF_num_types
        xdmf_num_types(1) = "Int"
        xdmf_num_types(2) = "Float"

        ! XDMF_format_types
        xdmf_format_types(1) = "XML"
        xdmf_format_types(2) = "HD5"

        ! XDMF_geom_types
        xdmf_geom_types(1) = "X_Y"
        xdmf_geom_types(2) = "X_Y_Z"

        ! XDMF_top_types
        xdmf_top_types(1) = "2DSMesh"
        xdmf_top_types(2) = "3DSMesh"

        ! XDMF_att_types
        xdmf_att_types(1) = "Scalar"
        xdmf_att_types(2) = "Vector"

        ! XDMF_center_types
        xdmf_center_types(1) = "Node"
        xdmf_center_types(2) = "Cell"

        ! XDMF_grid_types
        xdmf_grid_types(1) = "None"
        xdmf_grid_types(2) = "Temporal"
        xdmf_grid_types(3) = "Spatial"
    end subroutine init_hdf5

    !> \private array version
    subroutine dealloc_xml_str_arr(XML_str)
        ! input / output
        type(xml_str_type), intent(inout), allocatable :: XML_str(:)            !> array of XML strings to be deallocated

        ! local variables
        integer :: id                                                           ! counter

        ! deallocate individual arrays
        do id = 1,size(xml_str)
            call dealloc_xml_str_ind(xml_str(id))
        end do

        ! deallocate the array
        deallocate(xml_str)
    end subroutine dealloc_xml_str_arr
    !> \private individual version
    subroutine dealloc_xml_str_ind(XML_str)
        ! input / output
        type(xml_str_type), intent(out) :: XML_str                              !< XML string to be deallocated
    end subroutine dealloc_xml_str_ind

    !> \private rank 2 array version
    subroutine dealloc_var_1d_arr_2(var_1D)
        ! input / output
        type(var_1d_type), intent(inout), allocatable :: var_1D(:,:)            !< array of 1D variables to be deallocated

        ! local variables
        integer :: id, jd                                                       ! counters

        ! deallocate individual arrays
        do jd = 1,size(var_1d,2)
            do id = 1,size(var_1d,1)
                call dealloc_var_1d_ind(var_1d(id,jd))
            end do
        end do

        ! deallocate the array
        deallocate(var_1d)
    end subroutine dealloc_var_1d_arr_2
    !> \private array version
    subroutine dealloc_var_1d_arr(var_1D)
        ! input / output
        type(var_1d_type), intent(inout), allocatable :: var_1D(:)              !< array of 1D variables to be deallocated

        ! local variables
        integer :: id                                                           ! counter

        ! deallocate individual arrays
        do id = 1,size(var_1d)
            call dealloc_var_1d_ind(var_1d(id))
        end do

        ! deallocate the array
        deallocate(var_1d)
    end subroutine dealloc_var_1d_arr
```

```
    !> \private individual version
    subroutine dealloc_var_1d_ind(var_1D)
        ! input / output
        type(var_1d_type), intent(out) :: var_1D                                !< 1D variable to be deallocated
    end subroutine dealloc_var_1d_ind
end module hdf5_vars
```

D.40 /Users/toonweyens/Code/PB3D/Modules/HELENA_ops.f90 File Reference

Modules

- module helena_ops

  Operations on HELENA variables.

Functions/Subroutines

- integer function, public helena_ops::read_hel (n_r_in, use_pol_flux_h)

  Reads the HELENA equilibrium data.
- integer function, public helena_ops::interp_hel_on_grid (grid_in, grid_out, eq_2, x_1, x_2, eq_2_out, x↩_1_out, x_2_out, eq_1, grid_name)

  Interpolate variables resulting from HELENA equilibria to another grid (angularly).
- integer function, public helena_ops::test_metrics_h ()

  Checks whether the metric elements provided by HELENA are consistent with a direct calculation using the coordinate transformations [15].
- integer function, public helena_ops::test_harm_cont_h ()

  Investaige harmonic content of the HELENA variables.

D.41 HELENA_ops.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Operations on HELENA variables.
!-----------------------------------------------------------------------------!
module helena_ops
#include <PB3D_macros.h>
    use num_vars, only: pi
    use str_utilities
    use output_ops
    use messages
    use num_vars, only: dp, max_str_ln
    use grid_vars, only: grid_type
    use eq_vars, only: eq_1_type, eq_2_type
    use x_vars, only: x_1_type, x_2_type
    use helena_vars

    implicit none
    private
    public read_hel, interp_hel_on_grid
#if ldebug
    public test_metrics_h, test_harm_cont_h
#endif

contains
    !> Reads the HELENA equilibrium data
    !!
    !! Adapted from HELENA routine \c IODSK.
    !!
    !! The variales in the HELENA mapping file are globalized in two ways:
```

```
    !!  - X and  Y are normalized w.r.t. vacuum geometric  axis \c R_vac and
    !!  toroidal field at the geometric axis \c B_vac.
    !!  - <tt> R[m] = R_vac[m] (1 + eps X[]) </tt>,
    !!  - <tt> Z[m] = R_vac[m] eps Y[] </tt>.
    !!
    !! The covariant  toroidal field  \c F_H,  \c pres_H  and poloidal  flux are
    !! normalized  w.r.t  magnetic axis  \c  R_m  and  total toroidal  field  at
    !! magnetic axis \c B_m:
    !!  - <tt> RBphi[Tm]     = F_H[] R_m[m] B_m[T] </tt>,
    !!  - <tt> pres[N/m^2]   = pres_H[] (B_m[T])^2/mu_0[N/A^2] </tt>,
    !!  - <tt> flux_p[Tm^2]  = 2pi (s[])^2 cpsurf[] B_m[T] (R_m[m])^2 </tt>.
    !!
    !! The first  normalization type  is the  HELENA normalization,  whereas the
    !! second is the MISHKA normalization.
    !!
    !! Everything is translated to MISHKA  normalization to make comparison with
    !! MISHKA simple. This is done using the factors:
    !!  - <tt> radius[] = a[m] / R_m[m] </tt>,
    !!  - <tt> eps[] = a[m] / R_vac[m] </tt>,
    !! so that the expressions become:
    !!  - <tt> R[m]          = radius[] (1/eps[] + X[])          R_m[m]</tt>,
    !!  - <tt> Z[m]          = radius[] Y[]                      R_m[m]</tt>,
    !!  - <tt> RBphi[Tm]     = F_H[]                     B_m[T]  R_m[m]</tt>,
    !!  - <tt> pres[N/m^2]   = pres_H[]                  B_m[T]^2 mu_0[N/A^2]^-1</tt>,
    !!  - <tt> flux_p[Tm^2]  = 2pi (s[])^2 cpsurf[]      B_m[T]  R_m[T]^2</tt>.
    !!
    !! Finally, in  HELENA, the total  current \c I,  the poloidal beta  and the
    !! density at the geometric axis can be prescribed through:
    !!  - <tt> XIAB = mu_0 I / (a_vac B_vac) </tt>,
    !!  - <tt> BETAP = 8 pi S \<p\> / (I^2 mu_0) </tt>,
    !!  - <tt> ZN </tt>,
    !!
    !! where <tt> a_vac = eps R_vac </tt> and \c B_vac are vacuum quantities,
    !! \c  S is  the 2-D  cross-sectional  area and  \<p\> is  the 2-D  averaged
    !! pressure.
    !!
    !! \note To translate this to the MISHKA normalization factors as
    !!  - <tt> R_m = (eps/radius) R_vac </tt>,
    !!  - <tt> B_m = B_vac / B0 </tt>,
    !!
    !! \note where
    !!  - \c radius is in the mapping file (12), as well as in the HELENA output
    !!  (20).
    !!  - \c  eps is in the  mapping file (12), as  well as in the  HELENA input
    !!  (10) and output (20).
    !!  - \c  B0 is in the  mapping file (12), as  well as in the  HELENA output
    !!  (20).
    !! \note Furthermore,  the density on axis  can be specified as  \c ZN0 from
    !! HELENA input (10). \n The other  variables should probably not be touched
    !! for consistency.
    !! \note Finally, the variables IAS and B0 are in the mapping file (12) only
    !! in patched versions. See \ref tutorial_eq.
    !!
    !! \return ierr
    integer function read_hel(n_r_in,use_pol_flux_H) result(ierr)
        use num_vars, only: eq_name, eq_i, max_deriv, tol_zero, &
            &invert_top_bottom_h
        use num_utilities, only: calc_int, spline
        use helena_vars, only: pres_h, q_saf_h, rot_t_h, flux_p_h, flux_t_h, &
            &nchi, chi_h, ias, rbphi_h, r_h, z_h, rmtog_h, bmtog_h

        character(*), parameter :: rout_name = 'read_HEL'

        ! input / output
        integer, intent(inout) :: n_r_in                                    !< nr. of normal points in input grid
        logical, intent(inout) :: use_pol_flux_h                            !< .true. if HELENA equilibrium is based on pol. flux

        ! local variables
        character(len=max_str_ln) :: err_msg                                ! error message
        integer :: id, kd                                                   ! counters
        integer :: nchi_loc                                                 ! local nchi
        integer :: max_loc_z(2)                                             ! location of maximum Z
        real(dp), allocatable :: s_h(:)                                     ! flux coordinate s
        real(dp), allocatable :: curj(:)                                    ! toroidal current
        real(dp), allocatable :: vx(:), vy(:)                               ! R and Z of surface
        real(dp) :: diff_dp                                                 ! Dp difference
        real(dp) :: tol_diff_dp                                             ! tolerance on diff_Dp
        real(dp) :: dp0, dpe                                                ! derivative of pressure on axis and surface
        real(dp) :: dq0, dqe                                                ! derivative of safety factor on axis and surface
        real(dp) :: dj0, dje                                                ! derivative of toroidal current on axis and surface
        real(dp) :: df0, dfe                                                ! derivative of RBphi on axis and surface
        real(dp) :: radius, raxis                                           ! minor radius, major radius normalized w.r.t.  magnetic axis (i.e. R_m)
        real(dp) :: b0                                                      ! B_geo/B_m
```

```
        real(dp) :: eps                                                 ! inverse aspect ratio
        real(dp) :: cpsurf                                              ! poloidal flux on surface
        real(dp) :: ellip                                               ! ellipticity
        real(dp) :: tria                                                ! triangularity
        real(dp) :: delta_rz(2)                                         ! R and Z range
#if ldebug
        real(dp), allocatable :: inv_loc(:)                             ! used to invert top-bottom
#endif

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Reading data from HELENA output "' &
            &// trim(eq_name) // '"')
        call lvl_ud(1)

        ! set error messages
        err_msg = 'Failed to read HELENA output file. &
            &Does it output IAS and B0? See tutorial!'

        ! rewind input file
        rewind(eq_i)

        ! read mapped equilibrium from disk
        read(eq_i,*,iostat=ierr) n_r_in,ias                             ! nr. normal points (JS0), top-bottom symmetry
        chckerr(err_msg)
        n_r_in = n_r_in + 1                                             ! HELENA outputs nr. of normal points - 1

        allocate(s_h(n_r_in))                                           ! flux coordinate
        read(eq_i,*,iostat=ierr) (s_h(kd),kd=1,n_r_in)                  ! it is squared below, after reading cpsurf
        chckerr(err_msg)

        allocate(q_saf_h(n_r_in,0:max_deriv+1))                         ! safety factor
        read(eq_i,*,iostat=ierr) (q_saf_h(kd,0),kd=1,n_r_in)
        chckerr(err_msg)
        read(eq_i,*,iostat=ierr) dq0, dqe                               ! first point, last point
        chckerr(err_msg)
        ! Note: Dq0 an  Dqe are in the local coordinate  of the finite elements,
        ! and are therefore not used here: q_saf_H(:,1) will be overwritten
        q_saf_h(1,1) = dq0
        read(eq_i,*,iostat=ierr) (q_saf_h(kd,1),kd=2,n_r_in)            ! second to last point (again)
        chckerr(err_msg)

        allocate(curj(n_r_in))                                          ! toroidal current
        read(eq_i,*,iostat=ierr) (curj(kd),kd=1,n_r_in)
        chckerr(err_msg)

        read(eq_i,*,iostat=ierr) dj0,dje                                ! derivative of toroidal current at first and last point
        chckerr(err_msg)

        read(eq_i,*,iostat=ierr) nchi                                   ! nr. poloidal points
        chckerr(err_msg)
        nchi_loc = nchi
        if (ias.ne.0) nchi = nchi + 1                                   ! extend grid to 2pi if asymmetric (so doubling a point!)

        allocate(chi_h(nchi))                                           ! poloidal points
        read(eq_i,*,iostat=ierr) (chi_h(id),id=1,nchi_loc)
        chckerr(err_msg)
        if (ias.ne.0) chi_h(nchi) = 2*pi

        allocate(h_h_11(nchi,n_r_in))                                   ! upper metric factor 1,1
        read(eq_i,*,iostat=ierr) &
            &(h_h_11(mod(id-1,nchi_loc)+1,(id-1)/nchi_loc+1),&
            &id=nchi_loc+1,n_r_in*nchi_loc)                             ! (gem11)
        chckerr(err_msg)
        h_h_11(:,1) = tol_zero                                          ! first normal point is not given, so set to almost zero
        if (ias.ne.0) h_h_11(nchi,:) = h_h_11(1,:)

        allocate(h_h_12(nchi,n_r_in))                                   ! upper metric factor 1,2
        read(eq_i,*,iostat=ierr) &
            &(h_h_12(mod(id-1,nchi_loc)+1,(id-1)/nchi_loc+1),&
            &id=nchi_loc+1,n_r_in*nchi_loc)                             ! (gem12)
        chckerr(err_msg)
        do id = 1,nchi-1
            ierr = spline(s_h(2:n_r_in),h_h_12(id,2:n_r_in),s_h(1:1),&
                &h_h_12(id,1:1),deriv=0,extrap=.true.)
            chckerr('')
        end do
        if (ias.ne.0) h_h_12(nchi,:) = h_h_12(1,:)

        read(eq_i,*,iostat=ierr) cpsurf, radius                         ! poloidal flux on surface, minor radius
        chckerr(err_msg)
```

```fortran

        allocate(h_h_33(nchi,n_r_in))                                   ! lower metric factor 3,3
        read(eq_i,*,iostat=ierr) &
            &(h_h_33(mod(id-1,nchi_loc)+1,(id-1)/nchi_loc+1),id=&
            &nchi_loc+1,n_r_in*nchi_loc)                                ! (gem33)

        read(eq_i,*,iostat=ierr) raxis, b0                              ! major radius
        chckerr(err_msg)
        h_h_33(:,1) = raxis**2                                          ! first normal point is degenerate, major radius
        if (ias.ne.0) h_h_33(nchi,:) = h_h_33(1,:)
        h_h_33 = 1._dp/h_h_33                                           ! inverse is given

        allocate(pres_h(n_r_in,0:max_deriv+1))                          ! pressure profile
        read(eq_i,*,iostat=ierr) (pres_h(kd,0),kd=1,n_r_in)
        chckerr(err_msg)
        read(eq_i,*,iostat=ierr) dp0, dpe                               ! first point, last point
        chckerr(err_msg)

        allocate(rbphi_h(n_r_in,0:max_deriv+1))                         ! R B_phi (= F)
        read(eq_i,*,iostat=ierr) (rbphi_h(kd,0),kd=1,n_r_in)
        chckerr(err_msg)
        ! Note: DF0 and DFe contain rubbish and are not used.
        read(eq_i,*,iostat=ierr) df0, dfe                               ! derivatives of R B_phi on axis and surface
        chckerr(err_msg)

        allocate(vx(nchi))
        read(eq_i,*,iostat=ierr) (vx(id),id=1,nchi_loc)                 ! R on surface
        chckerr(err_msg)
        if (ias.ne.0) vx(nchi) = vx(1)

        allocate(vy(nchi))
        read(eq_i,*,iostat=ierr) (vy(id),id=1,nchi_loc)                 ! Z on surface
        chckerr(err_msg)
        if (ias.ne.0) vy(nchi) = vy(1)

        read(eq_i,*,iostat=ierr) eps                                    ! inverse aspect ratio
        chckerr(err_msg)

        allocate(r_h(nchi,n_r_in))                                      ! major radius R
        read(eq_i,*,iostat=ierr) (r_h(mod(id-1,nchi_loc)+1,(id-1)/nchi_loc+1),&
            &id=nchi_loc+1,n_r_in*nchi_loc)                             ! (xout)
        chckerr(err_msg)

        allocate(z_h(nchi,n_r_in))                                      ! height Z
        read(eq_i,*,iostat=ierr) (z_h(mod(id-1,nchi_loc)+1,(id-1)/nchi_loc+1),&
            &id=nchi_loc+1,n_r_in*nchi_loc)                             ! (yout)
        chckerr(err_msg)

        ! transform to MISHKA normalization
        allocate(flux_p_h(n_r_in,0:max_deriv+1))
        flux_p_h(:,0) = 2*pi*s_h**2 * cpsurf                            ! rescale flux coordinate (HELENA uses psi_pol/2pi as flux)
        radius = radius * raxis                                         ! global length normalization with R_m
        r_h = radius*(1._dp/eps + r_h)                                  ! local normalization with a
        z_h = radius*z_h                                                ! local normalization with a
        r_h(:,1) = raxis                                                ! first normal point is degenerate, major radius
        z_h(:,1) = 0._dp                                                ! first normal point is degenerate, 0
        if (ias.ne.0) r_h(nchi,:) = r_h(1,:)
        if (ias.ne.0) z_h(nchi,:) = z_h(1,:)
#if ldebug
        if (invert_top_bottom_h .and. ias.ne.0) then
            call print_ex_2d('boundary','RZ',z_h(:,n_r_in),&
                &x=r_h(:,n_r_in),draw=.false.)
            call draw_ex(['boundary'],'RZ',1,1,.false.)
            call writo('Inverting top and bottom to debug HELENA',&
                &warning=.true.)
            allocate(inv_loc(nchi))
            ! chi_H
            inv_loc = chi_h
            chi_h = 2*pi-inv_loc(nchi:1:-1)
            ! Z_H, R_H, h_H_11, h_H12, h_H_33
            do kd = 1,n_r_in
                inv_loc = z_h(:,kd)
                z_h(:,kd) = -inv_loc(nchi:1:-1)
                inv_loc = r_h(:,kd)
                r_h(:,kd) = inv_loc(nchi:1:-1)
                inv_loc = h_h_11(:,kd)
                h_h_11(:,kd) = inv_loc(nchi:1:-1)
                inv_loc = h_h_12(:,kd)
                h_h_12(:,kd) = -inv_loc(nchi:1:-1)
                inv_loc = h_h_33(:,kd)
                h_h_33(:,kd) = inv_loc(nchi:1:-1)
            end do
            deallocate(inv_loc)
            call print_ex_2d('boundary','RZ_inv_TB',z_h(:,n_r_in),&
```

```
                &x=r_h(:,n_r_in),draw=.false.)
            call draw_ex(['boundary'],'RZ_inv_TB',1,1,.false.)
        end if
#endif

        ! set conversion factors
        rmtog_h = radius/eps                                                    ! R_geo / R_mag
        bmtog_h = b0                                                            ! B_geo / B_mag

        ! calculate ellipticity and triangularity
        max_loc_z = maxloc(z_h)
        delta_rz(1) = maxval(r_h)-minval(r_h)
        if (ias.eq.0) then
            delta_rz(2) = 2*maxval(z_h)
        else
            delta_rz(2) = maxval(z_h)-minval(z_h)
        end if
        ellip = delta_rz(2)/delta_rz(1)
        tria = ((maxval(r_h)+minval(r_h))*0.5_dp-&
            &r_h(max_loc_z(1),max_loc_z(2)))/&
            &(delta_rz(2)*0.5_dp)
        call writo('Calculated ellipticity: '//trim(r2str(ellip)))
        call writo('Calculated triangularity: '//trim(r2str(tria)))

        ! calculate derivatives of fluxes
        allocate(flux_t_h(n_r_in,0:max_deriv+1))
        flux_p_h(:,1) = 2*pi
        flux_p_h(:,2:max_deriv+1) = 0._dp
        flux_t_h(:,1) = q_saf_h(:,0)*flux_p_h(:,1)
        ierr = calc_int(flux_t_h(:,1),flux_p_h(:,0)/(2*pi),flux_t_h(:,0))
        chckerr('')
        do kd = 2,max_deriv+1
            ierr = spline(flux_p_h(:,0)/(2*pi),flux_t_h(:,1),&
                &flux_p_h(:,0)/(2*pi),flux_t_h(:,kd),deriv=kd-1)
            chckerr('')
        end do

        ! calculate derivatives of other variables
        allocate(rot_t_h(n_r_in,0:max_deriv+1))                                 ! rotational transform
        rot_t_h(:,0) = 1._dp/q_saf_h(:,0)
        do kd = 1,max_deriv+1
            ierr = spline(flux_p_h(:,0)/(2*pi),pres_h(:,0),&
                &flux_p_h(:,0)/(2*pi),pres_h(:,kd),deriv=kd,bcs=[0,1],&
                &bcs_val=[0._dp,dpe*pi/(flux_p_h(n_r_in,0)*s_h(n_r_in))])
            chckerr('')
            ierr = spline(flux_p_h(:,0)/(2*pi),rbphi_h(:,0),&
                &flux_p_h(:,0)/(2*pi),rbphi_h(:,kd),deriv=kd)
            chckerr('')
            ierr = spline(flux_p_h(:,0)/(2*pi),rot_t_h(:,0),&
                &flux_p_h(:,0)/(2*pi),rot_t_h(:,kd),deriv=kd)
            chckerr('')
            ierr = spline(flux_p_h(:,0)/(2*pi),q_saf_h(:,0),&
                &flux_p_h(:,0)/(2*pi),q_saf_h(:,kd),deriv=kd)
            chckerr('')
        end do

        ! check for consistency
        tol_diff_dp = abs(pres_h(1,0)-pres_h(n_r_in,0))/n_r_in
        diff_dp = s_h(1)*flux_p_h(n_r_in,0)/pi*pres_h(1,1)-dp0
        if (abs(diff_dp).gt.tol_diff_dp) then
            call writo('Difference between pressure derivative on axis is &
                &large ('//trim(r2str(diff_dp))//')')
            call writo('Maybe increase precision or number of normal points',&
                &warning=.true.)
        end if
        diff_dp = s_h(n_r_in)*flux_p_h(n_r_in,0)/pi*pres_h(n_r_in,1)-dpe
        if (abs(diff_dp).gt.tol_diff_dp) then
            call writo('Difference between pressure derivative at edge is &
                &large ('//trim(r2str(diff_dp))//')')
            call writo('Maybe increase precision or number of normal points',&
                &warning=.true.)
        end if

        !!! To plot the cross-section
        !!call print_ex_2D(['cross_section'],'cross_section_H',Z_H,x=R_H,&
            !!&draw=.false.)
        !!call draw_ex(['cross_section'],'cross_section_H',n_r_in,1,.false.)
        !!call print_ex_2D(['cross_section_inv'],'cross_section_inv_H',&
            !!&transpose(Z_H),x=transpose(R_H),draw=.false.)
        !!call draw_ex(['cross_section_inv'],'cross_section_inv_H',nchi,1,.false.)

        ! HELENA always uses the poloidal flux
        use_pol_flux_h = .true.

        ! user output
        call writo('HELENA output given on '//trim(i2str(nchi))//&
            &' poloidal and '//trim(i2str(n_r_in))//' normal points')
```

```
        if (ias.eq.0) call writo('The equilibrium is top-bottom symmetric')
        call lvl_ud(-1)
        call writo('Data from HELENA output succesfully read')
    end function read_hel

    !> Calculate interpolation factors for  angular interpolation in \c grid_out
    !! of quantities defined on \c grid_in.
    !!
    !! This version is specific for  an input grid corresponding to axisymmetric
    !! variables with optional  top-down symmetry, as is the  case for variables
    !! resulting from HELENA equilibria.
    !!
    !! The  output  of  a 3-D  array  of  real  values  for the  poloidal  angle
    !! \f$\theta\f$ where the  floored integer of each value  indicates the base
    !! index of the interpolated value in the output grid and the modulus is the
    !! the fraction towards the next integer.
    !!
    !! The flag \c tb_sym indicates optionally that there is top-bottom symmetry
    !! as well as axisymmetry. When there is top-down symmetry, the variables in
    !! the lower  half (i.e. \f$-\pi  < \theta <  0\f$) are calculated  from the
    !! variables  in  the  upper  half  using the  symmetry  properties  of  the
    !! variables.  To indicate  this, the  sign of  the interpolation  factor is
    !! inverted to a negative value.
    !!
    !! The  displacement of  the theta  interval towards  the fundamental  theta
    !! interval is also outputted. For asymmetric variables this is for example:
    !!  -  1  for \f$-2\pi \ldots  0\f$
    !!  -  0  for \f$ 0    \ldots  2\pi\f$
    !!  - -1  for \f$ 2\pi \ldots  4\pi\f$
    !!
    !! etc.
    !!
    !! For symmetric variables, this is for example:
    !!  -  1  for \f$-3\pi \ldots -2\pi\f$,   with symmetry property
    !!  -  1  for \f$-2\pi \ldots -\pi\f$
    !!  -  0  for \f$-\pi  \ldots  0\f$,      with symmetry property
    !!  -  0  for \f$ 0    \ldots  \pi\f$
    !!  - -1  for \f$ \pi  \ldots  2\pi\f$,   with symmetry property
    !!  - -1  for \f$ 3\pi \ldots  3\pi\f$
    !! etc.
    !!
    !! By default,  the variables in the  Flux coord. system are  used, but this
    !! can be changed optionally with the flag \c "use_E.
    !!
    !! \return ierr
    integer function get_ang_interp_data_hel(grid_in,grid_out,theta_i,&
        &fund_theta_int_displ,tb_sym,use_E) result(ierr)
        use num_utilities, only: con2dis

        character(*), parameter :: rout_name = 'get_ang_interp_data_HEL'

        ! input / output
        type(grid_type), intent(in) :: grid_in                                  !< input grid
        type(grid_type), intent(in) :: grid_out                                 !< output grid
        real(dp), allocatable, intent(inout) :: theta_i(:,:,:)                  !< interpolation index
        integer, allocatable, intent(inout) :: fund_theta_int_displ(:,:,:)      !< displacement of fundamental theta interval
        logical, intent(in), optional :: tb_sym                                 !< top-bottom symmetry
        logical, intent(in), optional :: use_e                                  !< whether E is used instead of F

        ! local variables
        integer :: id, jd, kd                                                   ! counters
        logical :: tb_sym_loc                                                   ! local version of tb_sym
        logical :: use_e_loc                                                    ! local version of use_E
        real(dp) :: theta_loc                                                   ! local theta of output grid
        real(dp), pointer :: theta_in(:) => null()                              ! input theta
        character(len=max_str_ln) :: err_msg                                    ! error message
        integer :: fund_theta_int_lo, fund_theta_int_hi                         ! lower and upper limit of fundamental theta interval

        ! initialize ierr
        ierr = 0

        ! test whether axisymmetric grid
        if (grid_in%n(2).ne.1) then
            ierr = 1
            err_msg = 'Not an axisymmetric grid'
            chckerr(err_msg)
        end if
        ! test whether normal sizes compatible
        if (grid_in%loc_n_r.ne.grid_out%loc_n_r) then
            ierr = 1
            write(*,*) grid_in%loc_n_r, grid_out%loc_n_r
            err_msg = 'Grids are not compatible in normal direction'
            chckerr(err_msg)
        end if

```

```
        ! set up local use_E and tb_sym
        use_e_loc = .false.
        if (present(use_e)) use_e_loc = use_e
        tb_sym_loc = .false.
        if (present(tb_sym)) tb_sym_loc = tb_sym

        ! allocate theta_i and fundamental theta interval displacement
        allocate(theta_i(grid_out%n(1),grid_out%n(2),grid_out%loc_n_r))
        allocate(fund_theta_int_displ(grid_out%n(1),grid_out%n(2),&
            &grid_out%loc_n_r))

        ! initialize fundamental theta interval displacement
        fund_theta_int_displ = 0

        ! For every point on output grid, check to which half poloidal circle on
        ! the  axisymmetric input  grid  it  belongs to.  If  there is  top-down
        ! symmetry and the  angle theta lies in the bottom  part, the quantities
        ! have to be taken from their symmetric counterpart (2pi-theta).
        ! loop over all normal points of this rank
        do kd = 1,grid_out%loc_n_r
            ! point theta_in
            if (use_e) then
                theta_in => grid_in%theta_E(:,1,kd)                             ! axisymmetric grid should have only one toroidal point
            else
                theta_in => grid_in%theta_F(:,1,kd)                             ! axisymmetric grid should have only one toroidal point
            end if

            ! loop over all angles of ang_2
            do jd = 1,grid_out%n(2)
                ! loop over all angles of ang_1
                do id = 1,grid_out%n(1)
                    ! set the local poloidal point from theta
                    if (use_e) then
                        theta_loc = grid_out%theta_E(id,jd,kd)
                    else
                        theta_loc = grid_out%theta_F(id,jd,kd)
                    end if

                    ! set min
                    if (tb_sym) then
                        fund_theta_int_lo = -1
                        fund_theta_int_hi = 1
                    else
                        fund_theta_int_lo = 0
                        fund_theta_int_hi = 2
                    end if

                    ! add or subtract  2pi to the poloidal angle until  it is in
                    ! the fundamental  range indicated by  fund_theta_int_lo and
                    ! fund_theta_int_hi
                    if (theta_loc.lt.fund_theta_int_lo*pi) then
                        do while (theta_loc.lt.fund_theta_int_lo*pi)
                            theta_loc = theta_loc + 2*pi
                            fund_theta_int_displ(id,jd,kd) = &
                                &fund_theta_int_displ(id,jd,kd) - 1             ! theta interval lies to the left of fund. theta interval
                        end do
                    else if (theta_loc.gt.fund_theta_int_hi*pi) then
                        do while (theta_loc.gt.fund_theta_int_hi*pi)
                            theta_loc = theta_loc - 2*pi
                            fund_theta_int_displ(id,jd,kd) = &
                                &fund_theta_int_displ(id,jd,kd) + 1             ! theta interval lies to the right of fund. theta interval
                        end do
                    end if
                    ierr = con2dis(abs(theta_loc),theta_i(id,jd,kd),theta_in)
                    chckerr('')
                    if (theta_loc.lt.0) theta_i(id,jd,kd) = - theta_i(id,jd,kd)
                end do
            end do
        end do

        ! clean up
        nullify(theta_in)
    end function get_ang_interp_data_hel

    !> Interpolate  variables resulting from  HELENA equilibria to  another grid
    !! (angularly).
    !!
    !! The input  and  output grid  to be  provided  depend on  the
    !! quantities to be interpolated:
    !!   -  equilibrium  variables: flux  variables  (no  need to  convert)  and
    !!   derived quantities (need equilibrium grid)
    !!   - metric variables: jac_FD (need equilibrium grid)
    !!   - vectorial perturbation variables: U_i, DU_i (need perturbation grid)
```

```
    !!   - tensorial perturbation variables: PV_i, KV_i (need perturbation grid)
    !!
    !! Also, a message can be printed if a grid name is passed.
    !!
    !! \note
    !!   -# The  metric coefficients are  interpolated and then  compensated for
    !!   the straight-field-line coordinates as in \cite Weyens3D .
    !!   -# By default the interpolated  quantities overwrite the original ones,
    !!   but alternative output variables can be provided.
    !!   -#  as  the  equilibrium  and   perturbation  grid  are  not  generally
    !!   identical, this routine  has to be called separately  for the variables
    !!   tabulated in either grid.
    !!
    !! \return ierr
    integer function interp_hel_on_grid(grid_in,grid_out,eq_2,X_1,X_2,&
        &eq_2_out,X_1_out,X_2_out,eq_1,grid_name) result(ierr)

        use num_vars, only: use_pol_flux_f

        character(*), parameter :: rout_name = 'interp_HEL_on_grid'

        ! input / output
        type(grid_type), intent(in) :: grid_in                                  !< input grid
        type(grid_type), intent(in) :: grid_out                                 !< output grid
        type(eq_2_type), intent(inout), optional :: eq_2                        !< general metric equilibrium variables
        type(x_1_type), intent(inout), optional  :: x_1                         !< general vectorial perturbation variables
        type(x_2_type), intent(inout), optional  :: x_2                         !< general tensorial perturbation variables
        type(eq_2_type), intent(inout), optional :: eq_2_out                    !< field-aligned metric equilibrium variables
        type(x_1_type), intent(inout), optional  :: x_1_out                     !< field-aligned vectorial perturbation variables
        type(x_2_type), intent(inout), optional  :: x_2_out                     !< field-aligned tensorial perturbation variables
        type(eq_1_type), intent(in), optional :: eq_1                           !< general flux equilibrium variables for metric interpolation
        character(len=*), intent(in), optional :: grid_name                     !< name of grid to which to adapt quantities

        ! local variables
        real(dp), allocatable :: theta_i(:,:,:)                                 ! interpolation index
        integer, allocatable :: fund_theta_int_displ(:,:,:)                     ! displacement of fundamental theta interval
        logical :: tb_sym                                                       ! whether there is top-bottom symmetry (relevant only for HELENA)
        character(len=max_str_ln) :: err_msg                                    ! error message

        ! initialize ierr
        ierr = 0

        ! user output
        if (present(grid_name)) then
            call writo('Adapt quantities to '//trim(grid_name))
            call lvl_ud(1)
        end if

        ! tests
        if (present(eq_2).and..not.present(eq_1)) then
            ierr =1
            err_msg = 'To interpolate metric equilibrium variables, flux &
                &equilibrium has to be provided as well through eq_1'
            chckerr(err_msg)
        end if
        if (present(eq_2) .and. (present(x_1).or.present(x_2))) then
            ierr = 1
            err_msg = 'Only the quantities on one type of grid can be &
                &interpolated in a single call'
            chckerr(err_msg)
        end if

        ! set up tb_sym
        if (ias.eq.0) then
            tb_sym = .true.
        else
            tb_sym = .false.
        end if

        ! get angular interpolation factors
        ierr = get_ang_interp_data_hel(grid_in,grid_out,theta_i,&
            &fund_theta_int_displ,tb_sym=tb_sym,use_e=.false.)
        chckerr('')

        ! adapt equilibrium quantities
        if (present(eq_2)) then
            if (present(eq_2_out)) then
                ierr = interp_var_6d_real(eq_2%jac_FD,eq_2_out%jac_FD)
                chckerr('')
                ierr = interp_var_7d_real(eq_2%g_FD,eq_2_out%g_FD,sym_type=3)
                chckerr('')
                ierr = interp_var_7d_real(eq_2%h_FD,eq_2_out%h_FD,sym_type=4)
```

```
                chckerr('')
                ierr = interp_var_3d_real(eq_2%S,eq_2_out%S)
                chckerr('')
                ierr = interp_var_3d_real(eq_2%sigma,eq_2_out%sigma)
                chckerr('')
                ierr = interp_var_3d_real(eq_2%kappa_n,eq_2_out%kappa_n)
                chckerr('')
                ierr = interp_var_3d_real(eq_2%kappa_g,eq_2_out%kappa_g,&
                    &sym_type=2)
                chckerr('')
            else
                ierr = interp_var_6d_real_ow(eq_2%jac_FD)
                chckerr('')
                ierr = interp_var_7d_real_ow(eq_2%g_FD,sym_type=3)
                chckerr('')
                ierr = interp_var_7d_real_ow(eq_2%h_FD,sym_type=4)
                chckerr('')
                ierr = interp_var_3d_real_ow(eq_2%S)
                chckerr('')
                ierr = interp_var_3d_real_ow(eq_2%sigma)
                chckerr('')
                ierr = interp_var_3d_real_ow(eq_2%kappa_n)
                chckerr('')
                ierr = interp_var_3d_real_ow(eq_2%kappa_g,sym_type=2)
                chckerr('')
            end if
        end if

        ! adapt vectorial perturbation quantities
        if (present(x_1)) then
            if (present(x_1_out)) then
                ierr = interp_var_4d_complex(x_1%U_0,x_1_out%U_0,sym_type=2)
                chckerr('')
                ierr = interp_var_4d_complex(x_1%U_1,x_1_out%U_1,sym_type=2)
                chckerr('')
                ierr = interp_var_4d_complex(x_1%DU_0,x_1_out%DU_0,sym_type=1)
                chckerr('')
                ierr = interp_var_4d_complex(x_1%DU_1,x_1_out%DU_1,sym_type=1)
                chckerr('')
            else
                ierr = interp_var_4d_complex_ow(x_1%U_0,sym_type=2)
                chckerr('')
                ierr = interp_var_4d_complex_ow(x_1%U_1,sym_type=2)
                chckerr('')
                ierr = interp_var_4d_complex_ow(x_1%DU_0,sym_type=1)
                chckerr('')
                ierr = interp_var_4d_complex_ow(x_1%DU_1,sym_type=1)
                chckerr('')
            end if
        end if

        ! adapt tensorial perturbation quantities
        if (present(x_2)) then
            if (present(x_2_out)) then
                ierr = interp_var_4d_complex(x_2%PV_0,x_2_out%PV_0,sym_type=1)
                chckerr('')
                ierr = interp_var_4d_complex(x_2%PV_1,x_2_out%PV_1,sym_type=1)
                chckerr('')
                ierr = interp_var_4d_complex(x_2%PV_2,x_2_out%PV_2,sym_type=1)
                chckerr('')
                ierr = interp_var_4d_complex(x_2%KV_0,x_2_out%KV_0,sym_type=1)
                chckerr('')
                ierr = interp_var_4d_complex(x_2%KV_1,x_2_out%KV_1,sym_type=1)
                chckerr('')
                ierr = interp_var_4d_complex(x_2%KV_2,x_2_out%KV_2,sym_type=1)
                chckerr('')
            else
                ierr = interp_var_4d_complex_ow(x_2%PV_0,sym_type=1)
                chckerr('')
                ierr = interp_var_4d_complex_ow(x_2%PV_1,sym_type=1)
                chckerr('')
                ierr = interp_var_4d_complex_ow(x_2%PV_2,sym_type=1)
                chckerr('')
                ierr = interp_var_4d_complex_ow(x_2%KV_0,sym_type=1)
                chckerr('')
                ierr = interp_var_4d_complex_ow(x_2%KV_1,sym_type=1)
                chckerr('')
                ierr = interp_var_4d_complex_ow(x_2%KV_2,sym_type=1)
                chckerr('')
            end if
        end if

        ! user output
        if (present(grid_name)) then
            call lvl_ud(-1)
        end if
    contains
```

```
        ! Interpolate a  variable defined  on an  axisymmetric grid  at poloidal
        ! angles indicated by the interpolation factors theta_i.
        ! There  is  an  optional  variable   sym_type  that  allows  for  extra
        ! operations to be done on the variable if top-down symmetry is applied:
        !   - sym_type = 0: var(-theta) = var(theta)
        !   - sym_type = 1: var(-theta) = var(theta)*
        !   - sym_type = 2: var(-theta) = -var(theta)*
        !   - sym_type = 3: var(-theta) = +/-var(theta)* for g_FD (See note)
        !   - sym_type = 4: var(-theta) = +/-var(theta)* for h_FD (See note)
        ! where symmetry type 5 is only valid for tensorial quantities (7D). The
        ! transformation matrix, a function of  only the flux coordinate, has to
        ! be  passed as  well. This  is useful  for the  transformations of  the
        ! metric quantities as seen in [ADD REF].
        ! When top-down  symmetry has been  used to calculate  the interpolation
        ! factors, this is indicated by a  negative factor instead of a positive
        ! one.
        ! Note: This is also correct if i_hi = i_lo.
        ! Note:  For  the  overwrite  versions   ('ow'),  the  lower  limits  of
        ! indices are 0  for the dimensions corresponding  to derivatives. These
        ! allocations are done here and persist into the calling functions.
        ! Note: The  6D and 7D  overwrite versions take  a pointer instead  of a
        ! allocatable variable  because they  are to be  used for  jacobians and
        ! metric coefficients, which are defined as pointers.
        ! Note: symmetry type 3 for 7D variables (i.e. metric coefficients), the
        ! relation var(-theta)  = (-1)^(i+j) var(theta)*  where i and j  are the
        ! indices in g_ij  and h^ij. Furthermore, since  the metric coefficients
        ! are for  the Flux coordinate system,  they need to be  adapted for the
        ! secular coordinate theta or phi.
        !> \private
        integer function interp_var_3d_real(var,var_int,sym_type) result(ierr)  ! 3D_real version, separate output variable
            ! input / output
            real(dp), intent(in) :: var(1:,1:,1:)                               ! variable to be interpolated
            real(dp), intent(inout) :: var_int(1:,1:,1:)                        ! interpolated var
            integer, intent(in), optional :: sym_type                           ! optionally another type of symmetry

            ! local variables
            integer :: i_lo, i_hi                                               ! upper and lower index
            integer :: id, jd, kd                                               ! counters
            integer :: sym_type_loc                                             ! local version of symmetry type

            ! initialize ierr
            ierr = 0

            ! set up local symmetry type
            sym_type_loc = 0
            if (present(sym_type)) sym_type_loc = sym_type

            ! test symmetry types
            if (sym_type_loc.lt.0 .or. sym_type_loc.gt.2) then
                ierr = 1
                err_msg = 'Nonsensible symmetry type '//&
                    &trim(i2str(sym_type_loc))
                chckerr(err_msg)
            end if

            ! iterate over all normal points
            do kd = 1,size(var_int,3)
                ! iterate over all geodesical points
                do jd = 1,size(var_int,2)
                    ! iterate over all parallel points
                    do id = 1,size(var_int,1)
                        ! set up i_lo and i_hi
                        i_lo = floor(abs(theta_i(id,jd,kd)))
                        i_hi = ceiling(abs(theta_i(id,jd,kd)))

                        var_int(id,jd,kd) = var(i_lo,1,kd) + &
                            &(var(i_hi,1,kd)-var(i_lo,1,kd))*&
                            &(abs(theta_i(id,jd,kd))-i_lo)                      ! because i_hi - i_lo = 1
                        if (theta_i(id,jd,kd).lt.0) then
                            if (sym_type_loc.eq.2) var_int(id,jd,kd) = &
                                &- var_int(id,jd,kd)
                        end if
                    end do
                end do
            end do
        end function interp_var_3d_real
        !> \private
        integer function interp_var_3d_real_ow(var,sym_type) result(ierr)       ! 3D_real version, overwriting variable
            ! input / output
            real(dp), intent(inout), allocatable :: var(:,:,:)                  ! variable to be interpolated
            integer, intent(in), optional :: sym_type                           ! optionally another type of symmetry

            ! local variables
            real(dp), allocatable :: var_3d_loc(:,:,:)                          ! local 3D variable
            integer :: var_dim(3,2)                                             ! dimensions of variable

            ! initialize ierr
```

```
            ierr = 0

            ! set up variable dimensions
            var_dim(:,1) = [1,1,1]
            var_dim(:,2) = [shape(theta_i)]

            ! set up local 3D variable
            allocate(var_3d_loc(var_dim(1,1):var_dim(1,2),&
                &var_dim(2,1):var_dim(2,2),var_dim(3,1):var_dim(3,2)))

            ! call normal routine
            ierr = interp_var_3d_real(var,var_3d_loc,sym_type)
            chckerr('')

            ! reallocate variable
            deallocate(var)
            allocate(var(var_dim(1,1):var_dim(1,2),&
                &var_dim(2,1):var_dim(2,2),var_dim(3,1):var_dim(3,2)))

            ! overwrite variable
            var = var_3d_loc
        end function interp_var_3d_real_ow
        !> \private
        integer function interp_var_4d_complex(var,var_int,sym_type) &
            &result(ierr)                                                   ! 4D_complex version, separate output variable

            ! input / output
            complex(dp), intent(in) :: var(1:,1:,1:,1:)                     ! variable to be interpolated
            complex(dp), intent(inout) :: var_int(1:,1:,1:,1:)              ! interpolated var
            integer, intent(in), optional :: sym_type                       ! optionally another type of symmetry

            ! local variables
            integer :: i_lo, i_hi                                           ! upper and lower index
            integer :: id, jd, kd                                           ! counters
            integer :: sym_type_loc                                         ! local version of symmetry type

            ! initialize ierr
            ierr = 0

            ! set up local symmetry type
            sym_type_loc = 0
            if (present(sym_type)) sym_type_loc = sym_type

            ! test symmetry types
            if (sym_type_loc.lt.0 .or. sym_type_loc.gt.2) then
                ierr = 1
                err_msg = 'Nonsensible symmetry type '//&
                    &trim(i2str(sym_type_loc))
                chckerr(err_msg)
            end if

            ! iterate over all normal points
            do kd = 1,size(var_int,3)
                ! iterate over all geodesical points
                do jd = 1,size(var_int,2)
                    ! iterate over all parallel points
                    do id = 1,size(var_int,1)
                        ! set up i_lo and i_hi
                        i_lo = floor(abs(theta_i(id,jd,kd)))
                        i_hi = ceiling(abs(theta_i(id,jd,kd)))

                        var_int(id,jd,kd,:) = var(i_lo,1,kd,:) + &
                            &(var(i_hi,1,kd,:)-var(i_lo,1,kd,:))*&
                            &(abs(theta_i(id,jd,kd))-i_lo)                  ! because i_hi - i_lo = 1
                        if (theta_i(id,jd,kd).lt.0) then
                            if (sym_type_loc.eq.1) var_int(id,jd,kd,:) = &
                                &conjg(var_int(id,jd,kd,:))
                            if (sym_type_loc.eq.2) var_int(id,jd,kd,:) = &
                                &- conjg(var_int(id,jd,kd,:))
                        end if
                    end do
                end do
            end do
        end function interp_var_4d_complex
        !> \private
        integer function interp_var_4d_complex_ow(var,sym_type) result(ierr)    ! 4D_complex version, overwriting variable
            ! input / output
            complex(dp), intent(inout), allocatable :: var(:,:,:,:)         ! variable to be interpolated
            integer, intent(in), optional :: sym_type                       ! optionally another type of symmetry

            ! local variables
            complex(dp), allocatable :: var_4d_loc(:,:,:,:)                 ! local 4D variable
            integer :: var_dim(4,2)                                         ! dimensions of variable

            ! initialize ierr
            ierr = 0
```

```fortran

            ! set up variable dimensions
            var_dim(:,1) = [1,1,1,1]
            var_dim(:,2) = [shape(theta_i),size(var,4)]

            ! set up local 4D variable
            allocate(var_4d_loc(var_dim(1,1):var_dim(1,2),&
                &var_dim(2,1):var_dim(2,2),var_dim(3,1):var_dim(3,2),&
                &var_dim(4,1):var_dim(4,2)))

            ! call normal routine
            ierr = interp_var_4d_complex(var,var_4d_loc,sym_type)
            chckerr('')

            ! reallocate variable
            deallocate(var)
            allocate(var(var_dim(1,1):var_dim(1,2),&
                &var_dim(2,1):var_dim(2,2),var_dim(3,1):var_dim(3,2),&
                &var_dim(4,1):var_dim(4,2)))

            ! overwrite variable
            var = var_4d_loc
        end function interp_var_4d_complex_ow
        !> \private
        integer function interp_var_6d_real(var,var_int,sym_type) result(ierr)  ! 6D_real version, separate output variable
            ! input / output
            real(dp), intent(in) :: var(1:,1:,1:,0:,0:,0:)                      ! variable to be interpolated
            real(dp), intent(inout) :: var_int(1:,1:,1:,0:,0:,0:)               ! interpolated var
            integer, intent(in), optional :: sym_type                           ! optionally another type of symmetry

            ! local variables
            integer :: i_lo, i_hi                                               ! upper and lower index
            integer :: id, jd, kd, ld                                           ! counters
            integer :: sym_type_loc                                             ! local version of symmetry type

            ! initialize ierr
            ierr = 0

            ! set up local symmetry type
            sym_type_loc = 0
            if (present(sym_type)) sym_type_loc = sym_type

            ! test symmetry types
            if (sym_type_loc.lt.0 .or. sym_type_loc.gt.2) then
                ierr = 1
                err_msg = 'Nonsensible symmetry type '//&
                    &trim(i2str(sym_type_loc))
                chckerr(err_msg)
            end if

            ! iterate over all normal points
            do kd = 1,size(var_int,3)
                ! iterate over all geodesical points
                do jd = 1,size(var_int,2)
                    ! iterate over all parallel points
                    do id = 1,size(var_int,1)
                        ! set up i_lo and i_hi
                        i_lo = floor(abs(theta_i(id,jd,kd)))
                        i_hi = ceiling(abs(theta_i(id,jd,kd)))

                        var_int(id,jd,kd,:,:,:) = var(i_lo,1,kd,:,:,:) + &
                            &(var(i_hi,1,kd,:,:,:)-var(i_lo,1,kd,:,:,:))*&
                            &(abs(theta_i(id,jd,kd))-i_lo)                      ! because i_hi - i_lo = 1
                        if (theta_i(id,jd,kd).lt.0) then
                            if (sym_type_loc.eq.2) var_int(id,jd,kd,:,:,:) = &
                                &- var_int(id,jd,kd,:,:,:)
                            ! adapt the derivatives
                            if (use_pol_flux_f) then
                                do ld = 0,size(var_int,6)-1
                                    var_int(id,jd,kd,:,:,ld) = &
                                        &(-1)**ld*var_int(id,jd,kd,:,:,ld)
                                end do
                            else
                                ierr = 1
                                err_msg = '!!! INTERP_VAR_6D_REAL NOT YET &
                                    &IMPLEMENTED FOR TOROIDAL FLUX !!!'
                                chckerr(err_msg)
                            end if
                        end if
                    end do
                end do
            end do
        end function interp_var_6d_real
        !> \private
        integer function interp_var_6d_real_ow(var,sym_type) result(ierr)       ! 6D_real version, overwriting variable
            ! input / output
            real(dp), intent(inout), allocatable :: var(:,:,:,:,:,:)            ! variable to be interpolated
```

```
        integer, intent(in), optional :: sym_type                          ! optionally another type of symmetry

        ! local variables
        real(dp), allocatable :: var_6d_loc(:,:,:,:,:,:)                   ! local 6D variable
        integer :: var_dim(6,2)                                            ! dimensions of variable

        ! initialize ierr
        ierr = 0

        ! set up variable dimensions
        var_dim(:,1) = [1,1,1,0,0,0]
        var_dim(:,2) = [shape(theta_i),size(var,4)-1,size(var,5)-1,&
            &size(var,6)-1]

        ! set up local 6D variable
        allocate(var_6d_loc(var_dim(1,1):var_dim(1,2),&
            &var_dim(2,1):var_dim(2,2),var_dim(3,1):var_dim(3,2),&
            &var_dim(4,1):var_dim(4,2),var_dim(5,1):var_dim(5,2),&
            &var_dim(6,1):var_dim(6,2)))

        ! call normal routine
        ierr = interp_var_6d_real(var,var_6d_loc,sym_type)
        chckerr('')

        ! reallocate variable
        deallocate(var)
        allocate(var(var_dim(1,1):var_dim(1,2),&
            &var_dim(2,1):var_dim(2,2),var_dim(3,1):var_dim(3,2),&
            &var_dim(4,1):var_dim(4,2),var_dim(5,1):var_dim(5,2),&
            &var_dim(6,1):var_dim(6,2)))

        ! overwrite variable
        var = var_6d_loc
    end function interp_var_6d_real_ow
    !> \private
    integer function interp_var_7d_real(var,var_int,sym_type) result(ierr)  ! 7D_real version, separate output variable
        use num_utilities, only: calc_mult, c, add_arr_mult, derivs
        use num_vars, only: max_deriv

        ! input / output
        real(dp), intent(in) :: var(1:,1:,1:,1:,0:,0:,0:)                  ! variable to be interpolated
        real(dp), intent(inout) :: var_int(1:,1:,1:,1:,0:,0:,0:)           ! interpolated var
        integer, intent(in), optional :: sym_type                          ! optionally another type of symmetry

        ! local variables
        integer :: i_lo, i_hi                                              ! upper and lower index
        integer :: id, jd, kd, ld                                          ! counters
        integer :: sym_type_loc                                            ! local version of symmetry type
        real(dp), allocatable :: t_met(:,:,:,:,:,:)                        ! transformation matrix for metric elements
        integer, allocatable :: derivs_loc(:,:)                            ! array of all unique derivatives
        integer :: c_loc(2)                                                ! local c

        ! initialize ierr
        ierr = 0

        ! set up local symmetry type
        sym_type_loc = 0
        if (present(sym_type)) sym_type_loc = sym_type

        ! test symmetry types
        if (sym_type_loc.lt.0 .or. sym_type_loc.gt.4) then
            ierr = 1
            err_msg = 'Nonsensible symmetry type '//&
                &trim(i2str(sym_type_loc))
            chckerr(err_msg)
        end if

        ! interpolate
        do kd = 1,size(var_int,3)
            do jd = 1,size(var_int,2)
                do id = 1,size(var_int,1)
                    i_lo = floor(abs(theta_i(id,jd,kd)))
                    i_hi = ceiling(abs(theta_i(id,jd,kd)))

                    var_int(id,jd,kd,:,:,:,:) = var(i_lo,1,kd,:,:,:,:) + &
                        &(var(i_hi,1,kd,:,:,:,:)-var(i_lo,1,kd,:,:,:,:))*&
                        &(abs(theta_i(id,jd,kd))-i_lo)                     ! because i_hi - i_lo = 1
                end do
            end do
        end do

        ! inversion of poloidal coordinate if TBS and symmetry type 2, 3, 4
        if (sym_type_loc.eq.2 .or. sym_type_loc.eq.3 .or. &
            &sym_type_loc.eq.4) then
            do kd = 1,size(var_int,3)
                do jd = 1,size(var_int,2)
                    do id = 1,size(var_int,1)
```

```
                        if (theta_i(id,jd,kd).lt.0) then
                            ! invert value
                            if (sym_type_loc.eq.2) then
                                var_int(id,jd,kd,:,:,:,:) = &
                                    &- var_int(id,jd,kd,:,:,:,:)
                            else if (sym_type_loc.eq.3 .or. &
                                &sym_type_loc.eq.4) then                    ! metric coefficients: assuming 3D and symmetric
                                c_loc(1) = c([2,1],.true.)
                                c_loc(2) = c([3,2],.true.)
                                var_int(id,jd,kd,c_loc(1),:,:,:) = &
                                    &- var_int(id,jd,kd,c_loc(1),:,:,:)
                                var_int(id,jd,kd,c_loc(2),:,:,:) = &
                                    &- var_int(id,jd,kd,c_loc(2),:,:,:)
                            end if
                            ! invert derivatives
                            if (use_pol_flux_f) then                        ! invert parallel derivative
                                do ld = 0,size(var_int,7)-1
                                    var_int(id,jd,kd,:,:,:,ld) = (-1)**ld*&
                                        &var_int(id,jd,kd,:,:,:,ld)
                                end do
                            else
                                ierr = 1
                                err_msg = '!!! INTERP_VAR_7D_REAL NOT YET &
                                    &IMPLEMENTED FOR TOROIDAL FLUX !!!'
                                chckerr(err_msg)
                            end if
                        end if
                    end do
                end do
            end do
        end if

        ! set up displacement matrix if symmetry type 3 or 4
        if (sym_type_loc.eq.3 .or. sym_type_loc.eq.4) then
            allocate(t_met(size(var_int,1),size(var_int,2),&
                &size(var_int,3),0:size(var_int,5)-1,&
                &0:size(var_int,6)-1,0:size(var_int,7)-1))
            t_met = 0._dp
            if (use_pol_flux_f) then
                do ld = 0,max_deriv
                    do kd = 1,size(theta_i,3)
                        t_met(:,:,kd,0,ld,0) = &
                            &2*pi*eq_1%q_saf_FD(kd,ld+1)
                    end do
                end do
            else
                do ld = 0,max_deriv
                    do kd = 1,size(theta_i,3)
                        t_met(:,:,kd,0,ld,0) = &
                            &-2*pi*eq_1%rot_t_FD(kd,ld+1)
                    end do
                end do
            end if
            do kd = 1,size(theta_i,3)
                do jd = 1,size(theta_i,2)
                    do id = 1,size(theta_i,1)
                        t_met(id,jd,kd,:,:,:) = t_met(id,jd,kd,:,:,:)*&
                            &fund_theta_int_displ(id,jd,kd)
                    end do
                end do
            end do
        end if

        ! correct g_2i
        if (sym_type_loc.eq.3) then                                         ! lower metric elements h_FD
            if (use_pol_flux_f) then                                        ! poloidal flux coordinates
                do id = 0,max_deriv
                    derivs_loc = derivs(id)
                    do jd = 1,size(derivs_loc,2)
                        ! g_22 -> g_22 + T_met * g_32
                        c_loc(1) = c([3,2],.true.)
                        c_loc(2) = c([2,2],.true.)
                        ierr = add_arr_mult(&
                            &var_int(:,:,:,c_loc(1),:,:,:),t_met,&
                            &var_int(:,:,:,c_loc(2),&
                            &derivs_loc(1,jd),derivs_loc(2,jd),&
                            &derivs_loc(3,jd)),derivs_loc(:,jd))
                        chckerr('')
                        ! g_32 -> g_32 + T_met * g_31
                        c_loc(1) = c([3,1],.true.)
                        c_loc(2) = c([3,2],.true.)
                        ierr = add_arr_mult(&
                            &var_int(:,:,:,c_loc(1),:,:,:),t_met,&
                            &var_int(:,:,:,c_loc(2),&
                            &derivs_loc(1,jd),derivs_loc(2,jd),&
                            &derivs_loc(3,jd)),derivs_loc(:,jd))
```

```
                        chckerr('')
                        ! g_22 -> g_22 + T_met * g_32
                        c_loc(1) = c([3,2],.true.)
                        c_loc(2) = c([2,2],.true.)
                        ierr = add_arr_mult(&
                            &var_int(:,:,:,c_loc(1),:,:,:),t_met,&
                            &var_int(:,:,:,c_loc(2),&
                            &derivs_loc(1,jd),derivs_loc(2,jd),&
                            &derivs_loc(3,jd)),derivs_loc(:,jd))
                        chckerr('')
                        ! g_12 -> g_12 + T_met * g_11
                        c_loc(1) = c([1,1],.true.)
                        c_loc(2) = c([1,2],.true.)
                        ierr = add_arr_mult(&
                            &var_int(:,:,:,c_loc(1),:,:,:),t_met,&
                            &var_int(:,:,:,c_loc(2),&
                            &derivs_loc(1,jd),derivs_loc(2,jd),&
                            &derivs_loc(3,jd)),derivs_loc(:,jd))
                        chckerr('')
                    end do
                end do
            else                                                        ! toroidal flux coordinates
                ierr = 1
                err_msg = '!!! INTERP_VAR_7D_REAL NOT YET &
                    &IMPLEMENTED FOR TOROIDAL FLUX !!!'
                chckerr(err_msg)
            end if
        end if

        ! correct h_1j
        if (sym_type_loc.eq.4) then                                     ! upper metric elements h_FD
            if (use_pol_flux_f) then                                    ! poloidal flux coordinates
                do id = 0,max_deriv
                    derivs_loc = derivs(id)
                    do jd = 1,size(derivs_loc,2)
                        ! h^11 -> h^11 - T_met * h^12
                        c_loc(1) = c([1,2],.true.)
                        c_loc(2) = c([1,1],.true.)
                        ierr = add_arr_mult(&
                            &var_int(:,:,:,c_loc(1),:,:,:),-t_met,&
                            &var_int(:,:,:,c_loc(2),&
                            &derivs_loc(1,jd),derivs_loc(2,jd),&
                            &derivs_loc(3,jd)),derivs_loc(:,jd))
                        chckerr('')
                        ! h^12 -> h^12 - T_met * h^22
                        c_loc(1) = c([2,2],.true.)
                        c_loc(2) = c([1,2],.true.)
                        ierr = add_arr_mult(&
                            &var_int(:,:,:,c_loc(1),:,:,:),-t_met,&
                            &var_int(:,:,:,c_loc(2),&
                            &derivs_loc(1,jd),derivs_loc(2,jd),&
                            &derivs_loc(3,jd)),derivs_loc(:,jd))
                        chckerr('')
                        ! h^11 -> h^11 - T_met * h^12
                        c_loc(1) = c([1,2],.true.)
                        c_loc(2) = c([1,1],.true.)
                        ierr = add_arr_mult(&
                            &var_int(:,:,:,c_loc(1),:,:,:),-t_met,&
                            &var_int(:,:,:,c_loc(2),&
                            &derivs_loc(1,jd),derivs_loc(2,jd),&
                            &derivs_loc(3,jd)),derivs_loc(:,jd))
                        chckerr('')
                        ! h^13 -> h^13 - T_met * h^23
                        c_loc(1) = c([2,3],.true.)
                        c_loc(2) = c([1,3],.true.)
                        ierr = add_arr_mult(&
                            &var_int(:,:,:,c_loc(1),:,:,:),-t_met,&
                            &var_int(:,:,:,c_loc(2),&
                            &derivs_loc(1,jd),derivs_loc(2,jd),&
                            &derivs_loc(3,jd)),derivs_loc(:,jd))
                        chckerr('')
                    end do
                end do
            else                                                        ! toroidal flux coordinates
                ierr = 1
                err_msg = '!!! INTERP_VAR_7D_REAL NOT YET &
                    &IMPLEMENTED FOR TOROIDAL FLUX !!!'
                chckerr(err_msg)
            end if
        end if
    end function interp_var_7d_real
    !> \private
    integer function interp_var_7d_real_ow(var,sym_type) result(ierr)       ! 7D_real version, overwriting variable

        ! input / output
        real(dp), intent(inout), allocatable :: var(:,:,:,:,:,:,:)         ! variable to be interpolated
        integer, intent(in), optional :: sym_type                           ! optionally another type of symmetry
```

```

            ! local variables
            real(dp), allocatable :: var_7d_loc(:,:,:,:,:,:)                  ! local 7D variable
            integer :: var_dim(7,2)                                           ! dimensions of variable

            ! initialize ierr
            ierr = 0

            ! set up variable dimensions
            var_dim(:,1) = [1,1,1,1,0,0,0]
            var_dim(:,2) = [shape(theta_i),size(var,4),size(var,5)-1,&
                &size(var,6)-1,size(var,7)-1]

            ! set up local 7D variable
            allocate(var_7d_loc(var_dim(1,1):var_dim(1,2),&
                &var_dim(2,1):var_dim(2,2),var_dim(3,1):var_dim(3,2),&
                &var_dim(4,1):var_dim(4,2),var_dim(5,1):var_dim(5,2),&
                &var_dim(6,1):var_dim(6,2),var_dim(7,1):var_dim(7,2)))

            ! call normal routine
            ierr = interp_var_7d_real(var,var_7d_loc,sym_type)
            chckerr('')

            ! reallocate variable
            deallocate(var)
            allocate(var(var_dim(1,1):var_dim(1,2),&
                &var_dim(2,1):var_dim(2,2),var_dim(3,1):var_dim(3,2),&
                &var_dim(4,1):var_dim(4,2),var_dim(5,1):var_dim(5,2),&
                &var_dim(6,1):var_dim(6,2),var_dim(7,1):var_dim(7,2)))

            ! overwrite variable
            var = var_7d_loc
        end function interp_var_7d_real_ow
    end function interp_hel_on_grid

#if ldebug
    !> Checks whether the metric elements provided by HELENA are consistent with
    !! a direct calculation using the coordinate transformations \cite Weyens3D.
    !!
    !! Direct calculations used:
    !! \f[\begin{aligned}
    !!   \left|\nabla \psi\right|^2             &= \frac{1}{\mathcal{J}^2}
    !!      \left(\left(\frac{\partial Z}{\partial \chi}\right)^2 +
    !!      \left(\frac{\partial R}{\partial \chi}\right)^2\right) \\
    !!   \left|\nabla \psi \cdot \nabla \chi\right|   &= \frac{1}{\mathcal{J}^2}
    !!      \left(
    !!      \frac{\partial Z}{\partial \chi} \frac{\partial Z}{\partial \psi} -
    !!      \frac{\partial R}{\partial \chi} \frac{\partial R}{\partial \psi}
    !!      \right) \\
    !!   \left|\nabla \chi\right|^2             &= \frac{1}{\mathcal{J}^2}
    !!      \left(\left(\frac{\partial Z}{\partial \psi}\right)^2 +
    !!      \left(\frac{\partial R}{\partial \psi}\right)^2\right) \\
    !!   \left|\nabla \phi\right|^2             &= \frac{1}{R^2}
    !! \end{aligned}\f]
    !! with
    !! \f[\mathcal{J} =
    !!      \frac{\partial Z}{\partial \psi} \frac{\partial R}{\partial \chi} -
    !!      \frac{\partial R}{\partial \psi} \frac{\partial Z}{\partial \chi}\f]
    !!
    !! Also, test whether the pressure balance
    !! \f$\nabla p = \vec{J}\times\vec{B} \f$ is satisfied.
    !!
    !! \ldebug
    !!
    !! \return ierr
    integer function test_metrics_h() result(ierr)
        use num_vars, only: rank, norm_disc_prec_eq
        use output_ops, only: plot_diff_hdf5
        use input_utilities, only: get_int
        use grid_vars, only: n_r_eq
        use num_utilities, only: spline

        character(*), parameter :: rout_name = 'test_metrics_H'

        ! local variables
        integer :: id, kd                                                     ! counters
        integer :: bcs(2,3)                                                   ! boundary conditions (theta(even),
    theta(odd), r)
        real(dp) :: bcs_val(2,3)                                              ! values for boundary conditions
        real(dp), allocatable :: rchi(:,:), rpsi(:,:)                         ! chi and psi derivatives of R
        real(dp), allocatable :: zchi(:,:), zpsi(:,:)                         ! chi and psi derivatives of Z
        real(dp), allocatable :: jac(:,:)                                     ! jac as defined above
        real(dp), allocatable :: jac_alt(:,:)                                 ! alternative calculation for jac
        real(dp), allocatable :: h_h_11_alt(:,:,:)                            ! alternative calculation for upper metric
    factor 11
        real(dp), allocatable :: h_h_12_alt(:,:,:)                            ! alternative calculation for upper metric
    factor 12
```

```
        real(dp), allocatable :: h_h_33_alt(:,:,:)                          ! alternative calculation for upper metric factor 33
        character(len=max_str_ln) :: file_name                              ! name of plot file
        character(len=max_str_ln) :: description                            ! description of plot
        integer :: r_min = 4                                                ! first normal index that has meaning
        real(dp), allocatable :: tempvar(:,:,:,:)                           ! temporary variable

        ! initialize ierr
        ierr = 0

        if (rank.eq.0) then
            ! user output
            call writo('Checking consistency of metric factors')
            call lvl_ud(1)

            ! calculate  the  auxiliary quantities  Zchi,  zpsi,  Rchi and  Rpsi
            ! containing the derivatives as well as jac
            allocate(rchi(nchi,n_r_eq),rpsi(nchi,n_r_eq))
            allocate(zchi(nchi,n_r_eq),zpsi(nchi,n_r_eq))
            allocate(jac(nchi,n_r_eq))

            ! set up boundary conditions
            if (ias.eq.0) then                                              ! top-bottom symmetric
                bcs(:,1) = [1,1]                                            ! theta(even): zero first derivative
                bcs(:,2) = [2,2]                                            ! theta(odd): zero first derivative
            else
                bcs(:,1) = [-1,-1]                                          ! theta(even): periodic
                bcs(:,2) = [-1,-1]                                          ! theta(odd): periodic
            end if
            bcs(:,3) = [0,0]                                                ! r: not-a-knot
            bcs_val = 0._dp

            ! calculate derivatives
            do id = 1,nchi
                ierr = spline(flux_p_h(:,0)/(2*pi),r_h(id,:),&
                    &flux_p_h(:,0)/(2*pi),rpsi(id,:),ord=norm_disc_prec_eq,&
                    &deriv=1,bcs=bcs(:,3),bcs_val=bcs_val(:,3))
                chckerr('')
                ierr = spline(flux_p_h(:,0)/(2*pi),z_h(id,:),&
                    &flux_p_h(:,0)/(2*pi),zpsi(id,:),ord=norm_disc_prec_eq,&
                    &deriv=1,bcs=bcs(:,3),bcs_val=bcs_val(:,3))
                chckerr('')
            end do
            do kd = 1,n_r_eq
                ierr = spline(chi_h,r_h(:,kd),chi_h,rchi(:,kd),&
                    &ord=norm_disc_prec_eq,deriv=1,bcs=bcs(:,1),&
                    &bcs_val=bcs_val(:,1))                                  ! even
                chckerr('')
                ierr = spline(chi_h,z_h(:,kd),chi_h,zchi(:,kd),&
                    &ord=norm_disc_prec_eq,deriv=1,bcs=bcs(:,2),&
                    &bcs_val=bcs_val(:,2))                                  ! odd
                chckerr('')
            end do

            ! calculate Jacobian
            do kd = 1,n_r_eq
                jac(:,kd) = q_saf_h(kd,0)/(h_h_33(:,kd)*rbphi_h(kd,0))
            end do

            ! calculate Jacobian differently
            allocate(jac_alt(nchi,n_r_eq))
            jac_alt = r_h*(zchi*rpsi - zpsi*rchi)

            ! output jacobian
            ! set some variables
            file_name = 'TEST_jac_H'
            description = 'Testing whether the HELENA Jacobian is consistent.'

            ! plot difference
            call plot_diff_hdf5(&
                &reshape(jac(:,r_min:n_r_eq),&
                &[nchi,1,n_r_eq-r_min+1]),reshape(jac_alt(:,r_min:n_r_eq),&
                &[nchi,1,n_r_eq-r_min+1]),file_name,descr=description,&
                &output_message=.true.)

            ! calculate the metric factors directly
            allocate(h_h_11_alt(nchi,1,n_r_eq))
            allocate(h_h_12_alt(nchi,1,n_r_eq))
            allocate(h_h_33_alt(nchi,1,n_r_eq))

            h_h_11_alt(:,1,:) = (r_h/jac)**2 * (zchi**2 + rchi**2)
            h_h_12_alt(:,1,:) = -(r_h/jac)**2 * (zchi*zpsi + rchi*rpsi)
            h_h_33_alt(:,1,:) = 1._dp/(r_h**2)

            ! output h_H_11
            ! set some variables
            file_name = 'TEST_h_H_1_1'
```

```
            description = 'Testing whether the HELENA metric factor h_1_1 is &
                &consistent.'

            ! plot difference
            call plot_diff_hdf5(h_h_11_alt(:,:,r_min:n_r_eq),&
                &reshape(h_h_11(:,r_min:n_r_eq),[nchi,1,n_r_eq-r_min+1]),&
                &file_name,descr=description,output_message=.true.)

            ! output h_H_12
            ! set some variables
            file_name = 'TEST_h_H_1_2'
            description = 'Testing whether the HELENA metric factor h_1_2 is &
                &consistent.'

            ! plot difference
            call plot_diff_hdf5(h_h_12_alt(:,:,r_min:n_r_eq),&
                &reshape(h_h_12(:,r_min:n_r_eq),[nchi,1,n_r_eq-r_min+1]),&
                &file_name,descr=description,output_message=.true.)

            ! output h_H_33
            ! set some variables
            file_name = 'TEST_h_H_3_3'
            description = 'Testing whether the HELENA metric factor h_3_3 is &
                &consistent.'

            ! plot difference
            call plot_diff_hdf5(h_h_33_alt(:,:,r_min:n_r_eq),&
                &reshape(h_h_33(:,r_min:n_r_eq),[nchi,1,n_r_eq-r_min+1]),&
                &file_name,descr=description,output_message=.true.)

            ! user output
            call lvl_ud(-1)
            call writo('Test complete')

            ! user output
            call writo('Checking pressure balance')
            call lvl_ud(1)

            ! calculate auxiliary  quantities:
            !   1: D1 F ,
            !   2: D1 p ,
            !   3: D1 (q/F h_11) ,
            !   4: D2 (q/F h_12) .
            allocate(tempvar(nchi,1,n_r_eq,4))
            do kd = 1,n_r_eq
                tempvar(:,1,kd,1) = rbphi_h(kd,1)
                tempvar(:,1,kd,2) = pres_h(kd,1)
            end do
            do id = 1,nchi
                ierr = spline(flux_p_h(:,0)/(2*pi),&
                    &q_saf_h(:,0)/rbphi_h(:,0)*h_h_11(id,:),&
                    &flux_p_h(:,0)/(2*pi),tempvar(id,1,:,3),&
                    &ord=norm_disc_prec_eq,deriv=1,bcs=bcs(:,3),&
                    &bcs_val=bcs_val(:,3))
                chckerr('')
            end do
            do kd = 1,n_r_eq
                ierr = spline(chi_h,q_saf_h(kd,0)/rbphi_h(kd,0)*h_h_12(:,kd),&
                    &chi_h,tempvar(:,1,kd,4),ord=norm_disc_prec_eq,deriv=1,&
                    &bcs=bcs(:,2),bcs_val=bcs_val(:,2))                          ! odd
                chckerr('')
            end do

            ! calculate pressure  balance in tempvar(1)
            !   mu_0 p' = F/(qR^2) (d/d1 (h_11 q/F) + d/d2 (h_12 q/F) + q F')
            do kd = 1,n_r_eq
                tempvar(:,1,kd,1) = &
                    &-rbphi_h(kd,0)*h_h_33(:,kd)/q_saf_h(kd,0) * &
                    &(tempvar(:,1,kd,1)*q_saf_h(kd,0) + tempvar(:,1,kd,3) + &
                    &tempvar(:,1,kd,4))
            end do

            ! output difference with p'
            ! set some variables
            file_name = 'TEST_p_H'
            description = 'Testing whether the HELENA pressure balance is &
                &consistent'

            ! plot difference
            call plot_diff_hdf5(tempvar(:,:,:,1),tempvar(:,:,:,2),&
                &file_name,descr=description,output_message=.true.)

            ! user output
            call lvl_ud(-1)
            call writo('Test complete')
        end if
    end function test_metrics_h
```

```fortran

    !> Investaige harmonic content of the HELENA variables.
    !!
    !! \note Run with one process.
    integer function test_harm_cont_h() result(ierr)
        use grid_vars, only: n_r_eq
        use grid_utilities, only: nufft
        use helena_vars, only: nchi, chi_h, ias, r_h, z_h

        character(*), parameter :: rout_name = 'test_metrics_H'

        ! local variables
        integer :: kd                                                   ! counter
        integer :: nchi_full                                            ! nr. of pol. points in full grid, without doubling
        real(dp), allocatable :: f_loc(:,:)                             ! local Fourier modes
        real(dp), allocatable :: chi_full(:)                            ! chi_H in full poloidal grid
        real(dp), allocatable :: r_h_full(:,:)                          ! R_H in full poloidal grid
        real(dp), allocatable :: z_h_full(:,:)                          ! Z_H in full poloidal grid
        real(dp), allocatable :: r_h_f(:,:,:,:)                         ! Fourier modes of R_H
        real(dp), allocatable :: z_h_f(:,:,:,:)                         ! Fourier modes of Z_H
        character(len=max_str_ln) :: plot_name                          ! name of plot
        character(len=2) :: loc_str(2)                                  ! local string

        ! initialize ierr
        ierr = 0

        ! set up full R and Z
        if (ias.eq.0) then
            nchi_full = 2*(nchi-1)
            allocate(chi_full(nchi_full))
            allocate(r_h_full(nchi_full,n_r_eq))
            allocate(z_h_full(nchi_full,n_r_eq))
            chi_full(1:nchi-1) =              chi_h(1:nchi-1)
            chi_full(2*(nchi-1):nchi:-1) =    2*pi - chi_h(2:nchi)
            r_h_full(1:nchi-1,:) =            r_h(1:nchi-1,:)
            r_h_full(2*(nchi-1):nchi:-1,:) =  r_h(2:nchi,:)
            z_h_full(1:nchi-1,:) =            z_h(1:nchi-1,:)
            z_h_full(2*(nchi-1):nchi:-1,:) = -z_h(2:nchi,:)
        else
            nchi_full = nchi-1
            allocate(chi_full(nchi_full))
            allocate(r_h_full(nchi_full,n_r_eq))
            allocate(z_h_full(nchi_full,n_r_eq))
            chi_full(1:nchi_full)   = chi_h(1:nchi_full)
            r_h_full(1:nchi_full,:) = r_h(1:nchi_full,:)
            z_h_full(1:nchi_full,:) = z_h(1:nchi_full,:)
        end if

        ! calculate NUFFT
        do kd = 1,n_r_eq
            ierr = nufft(chi_full,r_h_full(:,kd),f_loc)
            chckerr('')
            if (kd.eq.1) allocate(r_h_f(size(f_loc,1),1,n_r_eq,2))
            r_h_f(:,1,kd,:) = f_loc
            ierr = nufft(chi_full,z_h_full(:,kd),f_loc)
            chckerr('')
            if (kd.eq.1) allocate(z_h_f(size(f_loc,1),1,n_r_eq,2))
            z_h_f(:,1,kd,:) = f_loc
        end do

        ! plot
        loc_str(1) = 'Re'
        loc_str(2) = 'Im'
        do kd = 1,2
            plot_name = loc_str(kd)//'_R_H_F'
            call print_ex_2d(['1'],plot_name,r_h_f(:,1,:,kd),draw=.false.)
            call draw_ex(['1'],plot_name,n_r_eq,1,.false.)
            plot_name = loc_str(kd)//'_Z_H_F'
            call print_ex_2d(['1'],plot_name,z_h_f(:,1,:,kd),draw=.false.)
            call draw_ex(['1'],plot_name,n_r_eq,1,.false.)

            plot_name = loc_str(kd)//'_R_H_F_log'
            call print_ex_2d(['1'],plot_name,&
                &log10(max(1.e-10_dp,abs(r_h_f(:,1,:,kd)))),&
                &draw=.false.)
            call draw_ex(['1'],plot_name,n_r_eq,1,.false.)
            plot_name = loc_str(kd)//'_Z_H_F_log'
            call print_ex_2d(['1'],plot_name,&
                &log10(max(1.e-10_dp,abs(z_h_f(:,1,:,kd)))),&
                &draw=.false.)
            call draw_ex(['1'],plot_name,n_r_eq,1,.false.)
        end do
        call plot_hdf5(['real','imag'],'R_H_F',r_h_f)
        call plot_hdf5(['real','imag'],'R_H_F_log',&
            &log10(max(1.e-10_dp,abs(r_h_f))))
        call plot_hdf5(['real','imag'],'Z_H_F',z_h_f)
```

```
        call plot_hdf5(['real','imag'],'Z_H_F_log',&
            &log10(max(1.e-10_dp,abs(z_h_f))))
    end function
#endif
end module helena_ops
```

D.42 /Users/toonweyens/Code/PB3D/Modules/HELENA_vars.f90 File Reference

Modules

- module helena_vars

  Variables that have to do with HELENA quantities.

Functions/Subroutines

- subroutine, public helena_vars::dealloc_hel

  Deallocates HELENA quantities that are not used any more.

Variables

- real(dp), public helena_vars::rmtog_h

  R_geo/R_mag.
- real(dp), public helena_vars::bmtog_h

  B_geo/B_mag.
- real(dp), dimension(:), allocatable, public helena_vars::chi_h

  poloidal angle
- real(dp), dimension(:,:), allocatable, public helena_vars::flux_p_h

  poloidal flux
- real(dp), dimension(:,:), allocatable, public helena_vars::flux_t_h

  toroidal flux
- real(dp), dimension(:,:), allocatable, public helena_vars::pres_h

  pressure profile
- real(dp), dimension(:,:), allocatable, public helena_vars::q_saf_h

  safety factor
- real(dp), dimension(:,:), allocatable, public helena_vars::rot_t_h

  rotational transform
- real(dp), dimension(:,:), allocatable, public helena_vars::rbphi_h

  $RB_\phi(=F)$
- real(dp), dimension(:,:), allocatable, public helena_vars::h_h_11

  upper metric factor h_{11} (gem11)
- real(dp), dimension(:,:), allocatable, public helena_vars::h_h_12

  upper metric factor h_{12} (gem12)
- real(dp), dimension(:,:), allocatable, public helena_vars::h_h_33

  upper metric factor h_{32} (1 / gem12)
- real(dp), dimension(:,:), allocatable, public helena_vars::r_h

  major radius R (xout)
- real(dp), dimension(:,:), allocatable, public helena_vars::z_h

  height Z (yout)
- integer, public helena_vars::nchi

  nr. of poloidal points
- integer, public helena_vars::ias

  0 if top-bottom symmetric, 1 if not

D.43 HELENA_vars.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Variables that have to do with HELENA quantities.
!-----------------------------------------------------------------------------!
module helena_vars
#include <PB3D_macros.h>
    use num_vars, only: pi
    use str_utilities
    use output_ops
    use messages
    use num_vars, only: dp, max_str_ln
    use grid_vars, only: grid_type
    use x_vars, only: x_1_type, x_2_type

    implicit none
    private
    public dealloc_hel, &
        &pres_h, q_saf_h, rot_t_h, flux_p_h, flux_t_h, nchi, chi_h, ias, &
        &rbphi_h, r_h, z_h, h_h_11, h_h_12, h_h_33, rmtog_h, bmtog_h

    ! global variables
    real(dp) :: rmtog_h                                                     !< R_geo/R_mag
    real(dp) :: bmtog_h                                                     !< B_geo/B_mag
    real(dp), allocatable :: chi_h(:)                                       !< poloidal angle
    real(dp), allocatable :: flux_p_h(:,:)                                  !< poloidal flux
    real(dp), allocatable :: flux_t_h(:,:)                                  !< toroidal flux
    real(dp), allocatable :: pres_h(:,:)                                    !< pressure profile
    real(dp), allocatable :: q_saf_h(:,:)                                   !< safety factor
    real(dp), allocatable :: rot_t_h(:,:)                                   !< rotational transform
    real(dp), allocatable :: rbphi_h(:,:)                                   !< \f$R B_\phi (= F) \f$
    real(dp), allocatable :: h_h_11(:,:)                                    !< upper metric factor \f$h_{11}\f$ (\c gem11)
    real(dp), allocatable :: h_h_12(:,:)                                    !< upper metric factor \f$h_{12}\f$ (\c gem12)
    real(dp), allocatable :: h_h_33(:,:)                                    !< upper metric factor \f$h_{32}\f$ (1 / \c gem12)
    real(dp), allocatable :: r_h(:,:)                                       !< major radius \f$R\f$ (xout)
    real(dp), allocatable :: z_h(:,:)                                       !< height \f$Z\f$ (yout)
    integer :: nchi                                                         !< nr. of poloidal points
    integer :: ias                                                          !< 0 if top-bottom symmetric, 1 if not

contains
    !> Deallocates HELENA quantities that are not used any more.
    subroutine dealloc_hel
#if ldebug
        use num_vars, only: rank, print_mem_usage

        ! local variables
        integer :: mem_diff                                                 ! difference in memory

        ! memory usage before deallocation
        if (print_mem_usage) mem_diff = get_mem_usage()
#endif

        ! deallocate
        deallocate(chi_h)
        deallocate(flux_p_h)
        deallocate(flux_t_h)
        deallocate(pres_h)
        deallocate(q_saf_h)
        deallocate(rot_t_h)
        deallocate(rbphi_h)
        deallocate(r_h)
        deallocate(z_h)
        deallocate(h_h_11)
        deallocate(h_h_12)
        deallocate(h_h_33)

#if ldebug
        ! memory usage difference after deallocation
        if (print_mem_usage) then
            mem_diff = mem_diff - get_mem_usage()
            call writo('Rank '//trim(i2str(rank))//' liberated '//&
                &trim(i2str(mem_diff))//'kB deallocating HELENA')
        end if
#endif
    end subroutine dealloc_hel
end module helena_vars
```

D.44 /Users/toonweyens/Code/PB3D/Modules/input_ops.f90 File Reference

Modules

- module input_ops

  Operations concerning giving input.

Functions/Subroutines

- integer function, public input_ops::read_input_opts ()

  Reads input options from user-provided input file.
- integer function, public input_ops::read_input_eq ()

  Reads the equilibrium input file if no Richardson restart.
- integer function, public input_ops::print_output_in (data_name, remove_previous_arrs)

  Print input quantities to an output file.

D.45 input_ops.f90

Go to the documentation of this file.

```
!-------------------------------------------------------------------------!
!> Operations concerning giving input.
!-------------------------------------------------------------------------!
module input_ops
#include <PB3D_macros.h>
    use str_utilities
    use output_ops
    use messages
    use num_vars, only: dp, max_str_ln

    implicit none
    private
    public read_input_opts, read_input_eq, print_output_in

contains
    !> Reads input options from user-provided input file.
    !!
    !! This  is done  by the  master process  and these  variables will  then be
    !! passed on through MPI to other processes in broadcast_input_opts().
    !!
    !! \return ierr
    integer function read_input_opts() result(ierr)
        use num_vars, only: &
            &max_it_zero, tol_zero, max_it_rich, input_i, use_pol_flux_f, &
            &ev_style, max_tot_mem, plot_resonance, tol_rich, &
            &n_sol_requested, rank, plot_magn_grid, plot_b, plot_j, &
            &plot_flux_q, plot_kappa, plot_sol_xi, plot_sol_q, plot_e_rec, &
            &use_normalization, n_sol_plotted, n_theta_plot, n_zeta_plot, &
            &ev_bc, rho_style, retain_all_sol, prog_style, norm_disc_prec_x, &
            &norm_disc_prec_eq, norm_disc_prec_sol, norm_disc_style_sol, &
            &bc_style, tol_norm, tol_slepc_loc => tol_slepc, max_it_slepc, &
            &plot_size, u_style, norm_style, x_style, matrix_slepc_style, &
            &input_name, rich_restart_lvl, eq_style, min_theta_plot, &
            &max_theta_plot, min_zeta_plot, max_zeta_plot, min_r_plot, &
            &max_r_plot, max_nr_backtracks_hh, post_style, plot_grid_style, &
            &magn_int_style, k_style, ex_plot_style, pert_mult_factor_post, &
            &sol_n_procs, n_procs, post_output_full, post_output_sol, &
            &ev_guess, solver_slepc_style, plot_vac_pot, min_rvac_plot, &
            &max_rvac_plot, min_zvac_plot, max_zvac_plot, n_vac_plot, &
            &alpha_style, x_grid_style, max_njq_change
        use eq_vars, only: rho_0, r_0, pres_0, b_0, psi_0, t_0
        use x_vars, only: min_r_sol, max_r_sol, n_mod_x, prim_x, min_sec_x, &
            &max_sec_x
        use rich_vars, only: rich_lvl, min_n_par_x, req_min_n_par_x
        use rich_ops, only: find_max_rich_lvl
        use grid_vars, only: n_r_sol, min_par_x, max_par_x, n_alpha, &
```

```
            &min_alpha, max_alpha

        character(*), parameter :: rout_name = 'read_input_opts'

        ! local variables
        character(len=max_str_ln) :: err_msg                                    !< error message
        integer :: pb3d_rich_lvl                                                !< Richardson level to post-process (for POST)
        integer, parameter :: max_size_tol_slepc = 100                          !< maximum size of tol_SLEPC
        real(dp) :: alpha                                                       !< field line label (for alpha_style 1)
        real(dp) :: tol_slepc(max_size_tol_slepc)                               !< tol_SLEPC
        real(dp), parameter :: min_tol = 1.e-12_dp                              !< minimum general tolerance
        real(dp), parameter :: max_tol = 1.e-3_dp                               !< maximum general tolerance

        ! input options
        namelist /inputdata_pb3d/ min_par_x, max_par_x, alpha, min_r_sol, &
            &max_r_sol, max_it_zero, tol_zero, use_pol_flux_f, rho_style, &
            &rho_0, plot_magn_grid, plot_b, plot_j, plot_flux_q, plot_kappa, &
            &prim_x, min_sec_x, max_sec_x, n_mod_x, use_normalization, &
            &n_theta_plot, n_zeta_plot, norm_disc_prec_eq, tol_norm, &
            &max_tot_mem, n_r_sol, max_it_rich, tol_rich, ev_style, ev_guess, &
            &plot_resonance, n_sol_requested, ev_bc, tol_slepc, sol_n_procs, &
            &retain_all_sol, pres_0, r_0, psi_0, b_0, t_0, norm_disc_prec_x, &
            &bc_style, max_it_slepc, norm_disc_prec_sol, norm_disc_style_sol, &
            &plot_size, u_style, norm_style, k_style, matrix_slepc_style, &
            &rich_restart_lvl, min_n_par_x, min_theta_plot, max_theta_plot, &
            &min_zeta_plot, max_zeta_plot, alpha_style, max_nr_backtracks_hh, &
            &magn_int_style, ex_plot_style, n_alpha, solver_slepc_style, &
            &x_grid_style, max_njq_change
        namelist /inputdata_post/ n_sol_plotted, n_theta_plot, n_zeta_plot, &
            &plot_resonance, plot_flux_q, plot_kappa, plot_magn_grid, plot_b, &
            &plot_j, plot_sol_xi, plot_sol_q, plot_e_rec, plot_size, &
            &pb3d_rich_lvl, max_it_zero, tol_zero, min_theta_plot, &
            &max_theta_plot, min_zeta_plot, max_zeta_plot, min_r_plot, &
            &max_r_plot, max_nr_backtracks_hh, post_style, plot_grid_style, &
            &max_tot_mem, ex_plot_style, plot_vac_pot, pert_mult_factor_post, &
            &min_rvac_plot, max_rvac_plot, min_zvac_plot, max_zvac_plot, &
            &n_vac_plot

        ! initialize ierr
        ierr = 0

        if (rank.eq.0) then                                                     ! only master
            call writo('Setting up user-provided input "'//trim(input_name)//&
                &'"')
            call lvl_ud(1)

            ! initialize input variables (optionally overwritten by user later)
            call writo('Initialize all the inputs with default values')
            call lvl_ud(1)

            ! common variables for all program styles
            sol_n_procs = n_procs                                               ! use equal amount of processes for solution as for rest of program
            max_tot_mem = 6000_dp                                               ! count with 6GB total
            plot_size = [10,5]                                                  ! size of plot in inch
            min_r_plot = 0._dp                                                  ! minimal plot normal range
            max_r_plot = 1._dp                                                  ! maximal plot normal range
            min_theta_plot = 1                                                  ! starting from pi gives nicer plots
            max_theta_plot = 3
            select case(eq_style)
                case (1)                                                        ! VMEC
                    n_theta_plot = 201                                          ! nr. poloidal points in plot
                    n_zeta_plot = 51                                            ! nr. toroidal points in plot
                    min_zeta_plot = 0                                           ! min. toroidal plot angle [pi]
                    max_zeta_plot = 2                                           ! max. toroidal plot angle [pi]
                case (2)                                                        ! HELENA
                    n_theta_plot = 501                                          ! nr. poloidal points in plot
                    n_zeta_plot = 1                                             ! nr. toroidal points in plot
                    min_zeta_plot = 0                                           ! min. toroidal plot angle [pi]
                    max_zeta_plot = min_zeta_plot                               ! max. toroidal plot angle [pi]
            end select
            plot_grid_style = 0                                                 ! normal plots on 3D geometry
            max_nr_backtracks_hh = 20                                           ! standard nr. of backtracks
            ex_plot_style = 1                                                   ! GNUPlot
            norm_disc_prec_eq = 3                                               ! cubic precision for normal discretization of equilibrium
            norm_disc_prec_x = 3                                                ! cubic precision for normal discretization of perturbation
            norm_disc_prec_sol = 3                                              ! cubic precision for normal discretization of solution

            ! select depending on program style
            select case (prog_style)
                case(1)                                                         ! PB3D
                    call default_input_pb3d()
                case(2)                                                         ! POST
```

```
                call default_input_post()
        end select

        call lvl_ud(-1)

        ! read user input
        ! select depending on program style
        select case (prog_style)
            case(1)                                                     ! PB3D
                read(unit=input_i,nml=inputdata_pb3d,iostat=ierr,&
                    &iomsg=err_msg)                                     ! read input data

                ! check input if successful read
                if (ierr.eq.0) then                                     ! input file succesfully read
                    call writo('Overwriting with user-provided file "'&
                        &//trim(input_name) // '"')

                    call writo('Checking user-provided file')

                    call lvl_ud(1)

                    ! adapt MPI variables if needed
                    call adapt_mpi()

                    ! adapt run-time variables if needed
                    ierr = adapt_run_pb3d()
                    chckerr('')

                    ! adapt plotting variables if needed
                    ierr = adapt_plot()
                    chckerr('')

                    ! adapt perturbation modes
                    ierr = adapt_x_modes()
                    chckerr('')

                    ! adapt min_n_par_X if needed
                    call adapt_min_n_par_x

                    ! adapt alpha if needed
                    ierr = adapt_alpha()
                    chckerr('')

                    ! adapt solution grid
                    ierr = adapt_sol_grid(min_r_sol,max_r_sol,'sol')
                    chckerr('')

                    ! adapt normalization variables if needed
                    ierr = adapt_normalization()
                    chckerr('')

                    ! adapt Richardson variables if needed
                    call adapt_rich

                    ! adapt input / output variables if needed
                    ierr = adapt_inoutput_pb3d()
                    chckerr('')

                    ! adapt Householder variables if needed
                    call adapt_zero

                    ! adapt tolerances if needed
                    call adapt_tols

                    call lvl_ud(-1)
                else                                                    ! cannot read input data
                    ierr = 1
                    call writo('Cannot read user-provided file "'&
                        &//trim(input_name)//'", error message:',&
                        &error=.true.)
                    call lvl_ud(1)
                    call writo('"'//trim(err_msg)//'"')
                    call lvl_ud(-1)
                    chckerr("")
                end if
            case(2)                                                     ! POST
                read(unit=input_i,nml=inputdata_post,iostat=ierr,&
                    &iomsg=err_msg)                                     ! read input data

                ! check input if successful read
                if (ierr.eq.0) then                                     ! input file succesfully read
                    call writo('Overwriting with user-provided file "'&
                        &//trim(input_name) // '"')

                    call writo('Checking user-provided file')

                    call lvl_ud(1)
```

```
                        ! adapt run-time variables if needed
                        ierr = adapt_run_post()
                        chckerr('')

                        ! adapt plotting variables if needed
                        ierr = adapt_plot()
                        chckerr('')

                        ! adapt plot grid and save to solution variable
                        ierr = adapt_sol_grid(min_r_plot,max_r_plot,'plot')
                        chckerr('')

                        ! adapt input / output variables if needed
                        ierr = adapt_inoutput_post()
                        chckerr('')

                        call lvl_ud(-1)
                    else                                                    ! cannot read input data
                        ierr = 1
                        call writo('Cannot read user-provided file "'&
                            &//trim(input_name)//'", error message:',&
                            &error=.true.)
                        call lvl_ud(1)
                        call writo('"'//trim(err_msg)//'"')
                        call lvl_ud(-1)
                        chckerr("")
                    end if
            end select

            ! user output
            call writo('Close input file')
            close(input_i)

            call lvl_ud(-1)
            call writo('Input values set')
        end if
    contains
        ! PB3D version
        !> \private
        subroutine default_input_pb3d
            !use num_vars, only: use_pol_flux_E

            ! concerning solution
            n_r_sol = 100                                                   ! 100 points in solution grid
            min_r_sol = 0.1_dp                                              ! minimum normal range
            max_r_sol = 1.0_dp                                              ! maximum normal range
            tol_norm = 0.05                                                 ! tolerance for normal range
            ev_style = 1                                                    ! slepc solver for EV problem

            ! concerning field line
            alpha = 0._dp                                                   ! field line based in outboard
            min_alpha = 0.0_dp                                              ! minimum field-line label [pi]
            max_alpha = 2.0_dp                                              ! maximum field-line label [pi]
            select case (eq_style)
                case (1)                                                    ! VMEC
                    alpha_style = 2                                         ! multiple field-lines, single turns
                    n_alpha = 10
                case (2)                                                    ! HELENA
                    alpha_style = 1                                         ! single field-line, multiple turns
            end select

            ! concerning perturbation
            min_n_par_x = req_min_n_par_x                                   ! nonsensible value to check for user overwriting
            min_par_x = 0.0_dp                                              ! minimum parallel angle [pi]
            max_par_x = 2.0_dp                                              ! maximum parallel angle [pi]
            prim_x = 20                                                     ! main mode number of perturbation
            min_sec_x = huge(1)                                             ! nonsensible value to check for user overwriting
            max_sec_x = huge(1)                                             ! nonsensible value to check for user overwriting
            n_mod_x = huge(1)                                               ! nonsensible value to check for user overwriting
            !use_pol_flux_F = use_pol_flux_E                                ! use same normal flux coordinate as the equilibrium
            use_pol_flux_f = .true.                                         ! use poloidail flux as toroidal flux is untested

            ! concerning normalization
            rho_0 = 1e-7_dp                                                 ! for fusion, particle density of around 5E19 for detuerium
            r_0 = huge(1._dp)                                               ! nonsensible value to check for user overwriting
            pres_0 = huge(1._dp)                                            ! nonsensible value to check for user overwriting
            psi_0 = huge(1._dp)                                             ! nonsensible value to check for user
```

```
overwriting
            b_0 = huge(1._dp)                                          ! nonsensible value to check for user
overwriting
            t_0 = huge(1._dp)                                          ! nonsensible value to check for user
overwriting

            ! concerning input / output
            n_sol_requested = 1                                        ! request solution closest to guess
            retain_all_sol = .false.                                   ! don't retain faulty ones
            plot_resonance = .false.                                   ! do not plot the q-profile with nq-m = 0
            plot_magn_grid = .false.                                   ! do not plot the magnetic grid
            plot_kappa = .false.                                       ! do not plot the curvature
            plot_b = .false.                                           ! do not plot the magnetic field
            plot_j = .false.                                           ! do not plot the current
            plot_flux_q = .false.                                      ! do not plot the flux quantities

            ! concerning Richardson extrapolation
            max_it_rich = 1                                            ! by default no Richardson extrapolation
            tol_rich = 1.e-4_dp                                        ! tolerance of 1E-4
            rich_restart_lvl = 1                                       ! don't restart

            ! concerning finding zeros
            max_it_zero = 100                                          ! maximum 100 iterations
            tol_zero = 1.0e-10_dp                                      ! very low tolerance for calculation of
field lines

            ! runtime variables
            use_normalization = .true.                                 ! use normalization for the variables
            norm_disc_style_sol = 2                                    ! left finite differences
            magn_int_style = 1                                         ! trapezoidal rule
            max_it_slepc = 1000                                        ! max. nr. of iterations for SLEPC
            ev_bc = 1._dp                                              ! use 1 as artificial EV for the Boundary
Conditions
            ev_guess = -0.3_dp                                         ! sensible guess
            tol_slepc = huge(1._dp)                                    ! nonsensible value to check for user
overwriting
            rho_style = 1                                              ! constant pressure profile, equal to rho_0
            u_style = 3                                                ! full expression for U, up to order 3
            k_style = 1                                                ! perpendicular kinetic energy normalized
            norm_style = 1                                             ! MISHKA normalization
            bc_style = [1,2]                                           ! left BC zeroed and right BC through
minimization of energy
            x_style = 2                                                ! fast style: mode numbers optimized in
normal coordinate
            solver_slepc_style = 1                                     ! Krylov-Schur
            matrix_slepc_style = 1                                     ! sparse matrix storage
            x_grid_style = huge(1)                                     ! nonsensible value to check for user
overwriting
            max_njq_change = 0.49_dp                                   ! maximum change of just under 0.5 for n q
(pol. flux) or n iota (tor. flux)
        end subroutine default_input_pb3d

        ! POST version
        !> \private
        subroutine default_input_post()
            ! concerning finding zeros
            max_it_zero = 200                                          ! more iterations than PB3D
            tol_zero = 1.0e-8_dp                                       ! less relative error than PB3D

            ! runtime variables
            plot_resonance = .true.                                    ! plot the q-profile with nq-m = 0
            plot_flux_q = .true.                                       ! plot the flux quantities
            plot_kappa = .true.                                        ! plot the curvature
            plot_magn_grid = .true.                                    ! plot the magnetic grid
            plot_b = .true.                                            ! plot the magnetic field
            plot_j = .true.                                            ! plot the current
            plot_sol_xi = .true.                                       ! plot plasma perturbation of solution
            plot_sol_q = .true.                                        ! plot magnetic perturbation of solution
            plot_e_rec = .true.                                        ! plot energy reconstruction
            plot_vac_pot = .true.                                      ! plot vacuum potential
            post_style = 1                                             ! process on extended plot grid
            write(*,*) '!!!!!! min and max need to take into account normalization'
            min_rvac_plot = 0.1*r_0                                    ! minimum R of vacuum plot
            max_rvac_plot = 2*r_0                                      ! maximum R of vacuum plot
            min_zvac_plot = -r_0                                       ! minimum R of vacuum plot
            max_zvac_plot = r_0                                        ! maximum R of vacuum plot
            n_vac_plot = [100,100]                                     ! number of points in R and Z of vacuum plot

            ! variables concerning input / output
            pert_mult_factor_post = 0._dp                              ! factor by which to XYZ is perturbed in
POST
            pb3d_rich_lvl = huge(1)                                    ! don't restart

            ! variables concerning output
            n_sol_plotted = -1                                         ! plot all solutions
        end subroutine default_input_post

```

```
        ! Checks  whether the  variables  concerning MPI  are chosen  correctly:
        !   sol_n_procs cannot be greater than n_procs. If it is lower than 1,
        !   the maximum amount of n_procs is used.
        !> \private
        subroutine adapt_mpi()
            use num_vars, only: n_procs

            if (sol_n_procs.lt.1) then
                call writo('Using the maximum number of MPI processes '//&
                    &trim(i2str(n_procs))//' for SLEPC')
                sol_n_procs = n_procs
            end if
            if (sol_n_procs.gt.n_procs) then
                call writo('Cannot use more than '//trim(i2str(n_procs))//&
                    &' MPI processes for SLEPC',warning=.true.)
                sol_n_procs = n_procs
            end if
        end subroutine adapt_mpi

        ! Checks whether the variables concerning run-time are chosen correctly:
        !   rho_style has  to be 1  (constant rho = rho_0),
        !   matrix_SLEPC_style has to be 0 or 1,
        !   max_it_SLEPC has to be at least 1,
        !   magnetic integral style has to be 1..2,
        !   perturbation grid style has to be 1..2,
        !   tensorial perturbation variables interpolation style has to be 1..2
        !   (for style 2, a warning should be displayed)
        !   for HELENA (eq_style 1), only poloidal flux can be used.
        !   norm_disc_prec_eq has to  be 3 because cubics are  the maximum order
        !   for splines  and derivatives of  order 3 are needed  for equilibrium
        !   quantities.
        !   The other  normal discretization  precisions can  be lower,  but not
        !   higher.
        !> \private
        integer function adapt_run_pb3d() result(ierr)
            use num_vars, only: eq_style

            character(*), parameter :: rout_name = 'adapt_run_PB3D'

            ! initialize ierr
            ierr = 0

            if (rho_style.ne.1) then
                ierr = 1
                err_msg = 'rho_style has to be 1 (constant)'
                chckerr(err_msg)
            end if
            if (eq_style.eq.2 .and. .not.use_normalization) then
                use_normalization = .true.
                call writo('normalization is always used for HELENA',&
                    &warning=.true.)
            end if
            if (matrix_slepc_style.lt.1 .or. matrix_slepc_style.gt.2) then
                ierr = 1
                err_msg = 'matrix_SLEPC_style has to be 1 (sparse) or 2 (shell)'
                chckerr(err_msg)
            end if
            if (max_it_slepc.le.0) then
                max_it_slepc = 1
                call writo('max_it_SLEPC has to be positive and is set to 1',&
                    &warning=.true.)
            end if
            if (magn_int_style.lt.1 .or. magn_int_style.gt.2) then
                ierr = 1
                err_msg = 'magn_int_style has to be 1 (trapezoidal) or 2 &
                    &(Simpson 3/8 rule)'
                chckerr(err_msg)
            end if
            if (eq_style.eq.2 .and. .not.use_pol_flux_f) then
                use_pol_flux_f = .true.
                call writo('can only use poloidal flux for HELENA',&
                    &warning=.true.)
            end if
            if (norm_disc_prec_eq.ne.3) then
                ierr =1
                err_msg = 'can only use norm_disc_prec_eq = 3 currently'
                chckerr(err_msg)
            end if
            if (norm_disc_prec_x.lt.1 .or. norm_disc_prec_x.gt.3) then
                ierr =1
                err_msg = 'norm_disc_prec_X has to be between 1 and 3'
                chckerr(err_msg)
            end if
            if (norm_disc_prec_sol.lt.1 .or. norm_disc_prec_sol.gt.3) then
                ierr = 1
                err_msg = 'norm_disc_prec_sol has to be between 1 and 3'
                chckerr(err_msg)
```

```
            end if
        end function adapt_run_pb3d

        ! Checks whether the variables concerning run-time are chosen correctly:
        !    POST_style  should be  1  (plotting  on extended  plot  grid) or  2
        !    (plotting on field-aligned grid also used in PB3D).
        !> \private
        integer function adapt_run_post() result(ierr)
            character(*), parameter :: rout_name = 'adapt_run_POST'

            ! initialize ierr
            ierr = 0

            if (post_style.lt.1 .or. post_style.gt.2) then
                ierr = 1
                err_msg = 'POST_style has to be 1 (extended grid) or 2 &
                    &(field-aligned grid)'
                chckerr(err_msg)
            end if
        end function adapt_run_post

        ! Checks whether the variables concerning output are chosen correctly:
        !   PB3D_rich_lvl can be at most the maximally found level
        !   POST_output_full is true if non-flux quantities are plot
        !   POST_output_sol is true if solution quantities are plot
        !> \private
        integer function adapt_inoutput_post() result(ierr)
            use num_vars, only: compare_tor_pos

            character(*), parameter :: rout_name = 'adapt_inoutput_POST'

            ! local variables
            integer :: max_pb3d_rich_lvl                                        ! maximum Richardson level

            ! initialize ierr
            ierr = 0

            ! find maximum levl
            ierr = find_max_rich_lvl(pb3d_rich_lvl,max_pb3d_rich_lvl)
            chckerr('')
            if (max_pb3d_rich_lvl.le.0) then
                call writo('No suitable Richardson level found, so only &
                    &equilibrium output will be done.',alert=.true.)
                plot_sol_xi = .false.
                plot_sol_q = .false.
                plot_e_rec = .false.
                pb3d_rich_lvl = 1
            else
                if (pb3d_rich_lvl.eq.huge(1)) then
                    pb3d_rich_lvl = max_pb3d_rich_lvl
                    call writo('Treating maximum Richardson level found: '//&
                        &trim(i2str(max_pb3d_rich_lvl)))
                else if (pb3d_rich_lvl.le.max_pb3d_rich_lvl) then
                    call writo('Treating Richardson level requested: '//&
                        &trim(i2str(pb3d_rich_lvl)))
                    if (pb3d_rich_lvl.lt.max_pb3d_rich_lvl) then
                        call lvl_ud(1)
                        call writo('Less than maximum Richardson level: '//&
                            &trim(i2str(max_pb3d_rich_lvl)),warning=.true.)
                        call lvl_ud(-1)
                    end if
                else if (pb3d_rich_lvl.gt.max_pb3d_rich_lvl) then
                    ierr = 1
                    err_msg = 'PB3D_rich_lvl cannot be higher than '//&
                        &trim(i2str(max_pb3d_rich_lvl))
                    chckerr(err_msg)
                else if (pb3d_rich_lvl.lt.1) then
                    call writo('Richardson level requested '//&
                        &trim(i2str(pb3d_rich_lvl))//', so only equilibrium &
                        &output will be done.',alert=.true.)
                    plot_sol_xi = .false.
                    plot_sol_q = .false.
                    plot_e_rec = .false.
                    pb3d_rich_lvl = 1
                end if
            end if
            max_it_rich = pb3d_rich_lvl
            rich_lvl = pb3d_rich_lvl

            ! set POST_output_full and POST_output_sol
            post_output_sol = plot_sol_xi .or. plot_sol_q .or. plot_e_rec
            post_output_full = post_output_sol .or. plot_b .or. plot_j .or. &
                &plot_kappa .or. plot_vac_pot .or. compare_tor_pos
        end function adapt_inoutput_post

        ! Checks whether the variables concerning output are chosen correctly:
        !   n_sol_requested has to be at least one,
```

```fortran
!   rich_restart_lvl can  be at most  one more than the  maximally found
!   level, nor can it be higher than max_it_rich.
!> \private
integer function adapt_inoutput_pb3d() result(ierr)
    character(*), parameter :: rout_name = 'adapt_inoutput_PB3D'

    ! initialize ierr
    ierr = 0

    if (n_sol_requested.lt.1) then
        n_sol_requested = 1
        call writo('n_sol_requested has been increased to '&
            &//trim(i2str(n_sol_requested)),warning=.true.)
    end if
    if (rich_restart_lvl.lt.1) then
        call writo('rich_restart_lvl was '//&
            &trim(i2str(rich_restart_lvl))//&
            &' but cannot be lower than 1, so it was reset to 1',&
            &warning=.true.)
        rich_restart_lvl = 1
    end if
    if (rich_restart_lvl.gt.max_it_rich) then
        ierr = 1
        err_msg = 'rich_restart_lvl not within 1..max_it_rich = 1..'//&
            &trim(i2str(max_it_rich))
        chckerr(err_msg)
    end if
    if (rich_restart_lvl.gt.1) then
        ierr = find_max_rich_lvl(pb3d_rich_lvl,pb3d_rich_lvl)
        chckerr('')
        if (rich_restart_lvl.gt.pb3d_rich_lvl+1) then
            ierr = 1
            if (pb3d_rich_lvl.gt.0) then
                err_msg = 'The highest Richardson level found was '//&
                    &trim(i2str(pb3d_rich_lvl))
            else
                err_msg = 'No Richardson level found'
            end if
            chckerr(err_msg)
        end if
    end if
end function adapt_inoutput_pb3d

! Checks whether the variables concerning plotting are chosen correctly:
!   n_theta_plot and n_zeta_plot have to be positive,
!   if n_theta_plot or  n_zeta_plot is equal to 1, the  minimum value is
!   chosen.
!   if  comparing different  toroidal positions,  only 2  values can  be
!   used.
!   ex_plot_style should be 1 (GNUPlot) or 2 (Bokeh / Mayavi)
!   if toroidal positions are compared, theta_plot limits should contain
!   0..2pi and max_r_plot should be 1.
!> \private
integer function adapt_plot() result(ierr)
    use num_vars, only: compare_tor_pos

    character(*), parameter :: rout_name = 'adapt_plot'

    ! local variables
    real(dp), parameter :: tol = 1.e-6_dp                           ! tolerance for checks
    character(len=max_str_ln) :: err_msg                            ! error message

    ! initialize ierr
    ierr = 0

    if (n_theta_plot.lt.1) then
        n_theta_plot = 1
        call writo('n_theta_plot has to be positive and is set to '//&
            &trim(i2str(n_theta_plot)),warning=.true.)
    end if
    if (n_zeta_plot.lt.1) then
        n_zeta_plot = 1
        call writo('n_zeta_plot has to be positive and is set to '//&
            &trim(i2str(n_zeta_plot)),warning=.true.)
    end if
    if (n_theta_plot.eq.1 .and. &
        &abs(max_theta_plot-min_theta_plot).gt.tol) then
        call writo('For n_theta_plot = 1, max_theta_plot is changed to &
            &min_theta_plot = '//trim(r2strt(min_theta_plot)),&
            &warning=.true.)
        max_theta_plot = min_theta_plot
    end if
    if (n_zeta_plot.eq.1 .and. &
        &abs(max_zeta_plot-min_zeta_plot).gt.tol) then
        call writo('For n_zeta_plot = 1, min_zeta_plot and &
            &max_zeta_plot are changed to their average = '//&
            &trim(r2strt((min_zeta_plot+max_zeta_plot)/2)),&
```

```
                    &warning=.true.)
                max_zeta_plot = 0.5*(min_zeta_plot+max_zeta_plot)
                min_zeta_plot = max_zeta_plot
            end if
            if (ex_plot_style.lt.1 .or. ex_plot_style.gt.2) then
                call writo('external plot style should be')
                call lvl_ud(1)
                call writo('1: GNUPlot')
                call writo('2: Bokeh (2-D) / Mayavi (3-D)')
                call lvl_ud(-1)
                call writo('Default (1: GNUPlot) chosen',warning=.true.)
                ex_plot_style = 1
            end if
            if (prog_style.eq.2) then                                       ! only for POST
                if (compare_tor_pos) then
                    if (n_zeta_plot.ne.3) then
                        ierr = 1
                        err_msg = 'For compare_tor_pos, you can only use &
                            &n_zeta = 3 (one point in the middle)'
                        chckerr(err_msg)
                    end if
                    if (post_style.ne.1) then
                        ierr = 1
                        err_msg = 'For compare_tor_pos, you can only use &
                            &POST_style = 1 (extended grid)'
                        chckerr(err_msg)
                    end if
                    if (min_r_plot.lt.tol_zero) then
                        ierr = 1
                        err_msg = 'min_r_plot has to be greater than 0.'
                        chckerr(err_msg)
                    end if
                    if (max_r_plot.lt.1._dp) then
                        ierr = 1
                        err_msg = 'max_r_plot has to be 1.'
                        chckerr(err_msg)
                    end if
                    if (abs(min_theta_plot-0).gt.tol_zero .or. &
                        &abs(max_theta_plot-2).gt.tol_zero) then
                        ierr = 1
                        err_msg = 'theta limits of plot need to be 0..2pi'
                        chckerr(err_msg)
                    end if
                end if
                if (min_rvac_plot.lt.0) then
                    call writo('The minimum value for Rvac_plot has to be &
                        &greater than zero',warning=.true.)
                    call lvl_ud(1)
                        min_rvac_plot = 0.1*r_0
                        call writo('It has been set to '//&
                            &trim(r2str(min_rvac_plot)))
                    call lvl_ud(-1)
                end if
                if (max_rvac_plot.lt.min_rvac_plot) then
                    ierr = 1
                    err_msg = 'The maximum value for Rvac_plot has to be &
                        &greater than the minimum value'
                    chckerr(err_msg)
                end if
                if (eq_style.eq.1 .and. plot_vac_pot) then
                    plot_vac_pot = .false.
                    call writo('Not plotting vacuum for VMEC yet',&
                        &warning=.true.)
                end if
            end if
        end function adapt_plot

        ! Checks whether  variables concerning  perturbation modes  are correct:
        !   the absolute value of prim_X has to be at least min_nm_X,
        !   deduces the X  style,
        !   min_sec_X  and  max_sec_X, pertaining  to  X  style 1  and  n_sec_X,
        !   pertaining to X style 2 cannot be both given,
        !   if  X  style  1  (prescribed),  max_sec_X has  to  be  greater  than
        !   min_sec_X in absolute value,
        !   if  X style 2 (fast), n_mod_X has to be a number greater than 0,
        !   for every X style, some checks are made and some variables set.
        !> \private
        integer function adapt_x_modes() result(ierr)
            use x_vars, only: min_nm_x

            character(*), parameter :: rout_name = 'adapt_X_modes'

            ! local variables
            character(len=max_str_ln) :: err_msg                            ! error message

            ! initialize ierr
            ierr = 0
```

```fortran

            ! check prim_X: common for all X styles
            if (abs(prim_x).lt.min_nm_x) then
                ierr = 1
                err_msg = 'prim_X should be at least '//trim(i2str(min_nm_x))//&
                    &', and probably a bit higher still'
                chckerr(err_msg)
            end if

            ! set X_style
            if (min_sec_x.ne.huge(1) .and. min_sec_x.ne.huge(1)) then           ! min_sec_X and max_sec_X prescribed
                if (n_mod_x.eq.huge(1)) then                                    ! n_mod_X not prescribed
                    x_style = 1                                                 ! prescribed style
                else                                                            ! n_mod_X also prescribed
                    ierr = 1
                    err_msg = 'Cannot prescribe both min and max of secondary &
                        &X and nr. of modes.'
                    chckerr(err_msg)
                end if
            else                                                                ! min_sec_X and max_sec_X not prescribed
                if (n_mod_x.eq.huge(1)) then                                    ! n_mod_X not prescribed either: default
                    x_style = 2                                                 ! X style 2: fast
                    n_mod_x = 10                                                ! 10 resonating modes at every place
                else                                                            ! but n_mod_X prescribed
                    x_style = 2
                end if
            end if

            ! set X_grid_style if not overwritten by user
            if (x_grid_style.ge.huge(1)) then
                select case (x_style)
                    case (1)                                                    ! prescribed
                        x_grid_style = 1                                        ! equilibrium
                    case (2)                                                    ! fast
                        x_grid_style = 3                                        ! enriched
                end select
            end if

            ! check X_grid style
            if (x_grid_style.lt.1 .or. x_grid_style.gt.3) then
                ierr = 1
                err_msg = 'X_grid_style has to be 1 (equilibrium), 2 &
                    &(solution) or 3 (enriched)'
                chckerr(err_msg)
            end if

            ! checks for each X style
            select case (x_style)
                case (1)                                                        ! prescribed
                    ! check min_sec_X, max_sec_X
                    if (abs(max_sec_x).lt.abs(min_sec_x)) then
                        ierr = 1
                        err_msg = 'max_sec_X has to be larger or equal to &
                            &min_sec_X in absolute value'
                        chckerr(err_msg)
                    end if
                    if (max_sec_x.ge.huge(1)) then
                        ierr = 1
                        err_msg = 'max_sec_X not set'
                        chckerr(err_msg)
                    end if
                    if (min_sec_x.ge.huge(1)) then
                        ierr = 1
                        err_msg = 'min_sec_X not set'
                        chckerr(err_msg)
                    end if
                    if (x_grid_style.eq.3) then
                        ierr = 1
                        err_msg = 'X_grid_style 3 (enriched) does not make &
                            &sense for X_style 1 (prescribed)'
                        chckerr(err_msg)
                    end if

                    ! set n_mod_X
                    n_mod_x = abs(max_sec_x)-(min_sec_x)+1
                case (2)                                                        ! fast
                    if (n_mod_x.lt.1) then
                        ierr = 1
                        err_msg = 'the number of resonating modes has to be at &
                            &least one'
                        chckerr(err_msg)
                    end if
            end select
        end function adapt_x_modes

        ! Checks whether  min_n_par_X has  been chosen  correctly:
        !   It should be larger than  req_min_n_par_X (arbitrary) if an integral
```

```
!   is to be calculated. However,  there are additional requirements for
!   the different magnetic integral styles: for style 1 min_par_X has to
!   be of the form 2+k and for style 2 of the form 4+3k.
!> \private
subroutine adapt_min_n_par_x
    ! local variables
    integer :: fund_n_par                                           ! fundamental interval width

    ! check req_min_n_par_X
    if (min_n_par_x.lt.req_min_n_par_x)  then
        min_n_par_x = req_min_n_par_x
        call writo('min_n_par_X has been increased to '//&
            &trim(i2str(min_n_par_x))//' to have enough parallel &
            &resolution',warning=.true.)
    end if

    ! check for individual magnetic integral style
    select case (magn_int_style)
        case (1)                                                    ! Trapezoidal rule
            fund_n_par = 1
        case (2)                                                    ! Simpson's 3/8 rule
            fund_n_par = 3
    end select
    if (mod(min_n_par_x-1,fund_n_par).ne.0) then                    ! there is a remainder
        min_n_par_x = 1 + fund_n_par * &
            &((min_n_par_x-1)/fund_n_par + 1)
        call writo('min_n_par_X has been increased to '//&
            &trim(i2str(min_n_par_x))//' to be a of the form '//&
            &trim(i2str(fund_n_par+1))//' + '//&
            &trim(i2str(fund_n_par))//&
            &'k for magnetic integral style '//&
            &trim(i2str(magn_int_style)),warning=.true.)
    end if
end subroutine adapt_min_n_par_x

! Checks whether variables related to alpha are chosen correctly :
!   alpha_style has to be either 0 or 1
!   for 1, n_alpha = 1 and min/max_alpha = alpha
!   for 2, max_par_X - min_par_X = 2
!> \private
integer function adapt_alpha() result(ierr)
    character(*), parameter :: rout_name = 'adapt_alpha'

    ! local variables
    character(len=max_str_ln) :: err_msg                            ! error message

    ! initialize ierr
    ierr = 0

    ! check alpha style
    if (alpha_style.lt.1 .or. alpha_style.gt.2) then
        ierr = 1
        err_msg = 'Alpha style needs to be 1 or 2 [def]'
        chckerr(err_msg)
    end if

    ! adapt variables pertaining to particular alpha style
    select case (alpha_style)
        case (1)                                                    ! single field line, multiple turns
            if (n_alpha.ne.1) then
                call writo('For alpha style 1, n_alpha can only be 1',&
                    &warning=.true.)
                n_alpha = 1
            end if
            min_alpha = alpha
            max_alpha = alpha
        case (2)                                                    ! multiple field lines, single turns
            if (n_alpha.lt.1) then
                ierr = 1
                err_msg = 'For alpha style 2, n_alpha needs to be &
                    &positive'
                chckerr(err_msg)
            end if
            if (abs(max_par_x - min_par_x - 2) .gt. tol_zero) then
                min_par_x = 0._dp
                max_par_x = 2._dp
                call writo('min/max_par_X has been set to ['//&
                    &trim(r2strt(min_par_x))//':'//&
                    &trim(r2strt(max_par_x))//']',warning=.true.)
            end if
    end select

    ! for HELENA, warn about axisymmetry
    if (eq_style.eq.2) then
        if (abs(max_par_x - min_par_x - 2) .gt. tol_zero) then
            call writo('HELENA equilibria are axisymmetric, so only &
                &max - min_par_X = 2 needed',warning=.true.)
```

```
            end if
            if (n_alpha .gt. 1) then
                call writo('HELENA equilibria are axisymmetric, so only &
                    &n_alpha = 1 needed',warning=.true.)
            end if
        end if
    end function adapt_alpha

    ! Checks  whether variables  concerning the  solution grid  are correct:
    !   min_r should be greater than 0
    !   max_r should be lesser than 1
    !   min_r should be lesser than max_r
    ! The number of solution points should be large enough for the numerical
    ! discretization scheme, and there should be enough points per process.
    !> \private
    integer function adapt_sol_grid(min_r,max_r,var_name) result(ierr)
        use num_vars, only: n_procs

        character(*), parameter :: rout_name = 'adapt_sol_grid'

        ! input / output
        real(dp), intent(inout) :: min_r, max_r                          ! min and max of norma range
        character(len=*), intent(in) :: var_name                         ! name of variable

        ! local variables
        character(len=max_str_ln) :: err_msg                             ! error message

        ! initialize ierr
        ierr = 0

        ! check min_r
        if (min_r.lt.0._dp) then
            min_r = 0._dp
            call writo('min_r_'//trim(var_name)//' has been increased to '&
                &//trim(r2strt(min_r)),warning=.true.)
        end if

        ! check if max_r is not greater than 1
        if (max_r.gt.1._dp) then
            max_r = 1._dp
            call writo('max_r_'//trim(var_name)//' has been decreased to '&
                &//trim(r2strt(max_r)),warning=.true.)
        end if

        ! check if min_r < max_r
        if (min_r.ge.max_r) then
            ierr = 1
            err_msg = 'min_r_'//trim(var_name)//' should be lower than &
                & max_r_'//trim(var_name)
            chckerr(err_msg)
        end if

        ! check n_r_sol
        if (n_r_sol.lt.6*norm_disc_prec_sol+2) then
            n_r_sol = 6*norm_disc_prec_sol+2
            call writo('n_r_sol has been increased to '//&
                &trim(i2str(n_r_sol))//' to have enough points for the &
                &normal discretization precision',warning=.true.)
        end if
        if (n_r_sol.lt.n_procs*(2*norm_disc_prec_sol+1)) then
            n_r_sol = n_procs*(2*norm_disc_prec_sol+1)
            call writo('n_r_sol has been increased to '//&
                &trim(i2str(n_r_sol))//' to have enough points per &
                &process',warning=.true.)
        end if

        ! warning if too close to axis for VMEC
        if (eq_style.eq.1) then
            if (min_r.le.tol_zero) &
                &call writo('For VMEC, you should not go all the way to &
                &the magnetic axis.',warning=.true.)
        end if
    end function adapt_sol_grid

    ! Checks  whether the variables concerning  Richardson extrapolation are
    ! correct:
    !   max_it_rich has to be at least 1.
    !> \private
    subroutine adapt_rich
        if (max_it_rich.lt.1) then
            max_it_rich = 1
            call writo('max_it_rich has been increased to 1',warning=.true.)
        end if
    end subroutine adapt_rich

    ! Checks whether the variables concerning finding zeros are correct:
    !   max_it_zero has to be at least 2,
```

```
        !> \private
        subroutine adapt_zero
            if (max_it_zero.lt.1) then
                max_it_zero = 2
                call writo('max_it_zero has been increased to 2',warning=.true.)
            end if
        end subroutine adapt_zero

        ! Checks whether tolerances are correct:
        !   tol_norm needs to be 0..1,
        !   tol_zero needs to be min_tol..max_tol,
        !   tol_rich needs to be min_tol..max_tol,
        !   tol_SLEPC needs to be min_tol..max_tol.
        ! Also sets local tol_SLEPC.
        !> \private
        subroutine adapt_tols
            ! local variables
            integer :: id                                                   ! counter
            integer :: max_id                                               ! maximum of user-set id
            real(dp), parameter :: tol_slepc_def = 1.e-5_dp                 ! default tol_SLEPC

            ! check tol_norm
            if (tol_norm.lt.0._dp) then
                call writo('tol_norm has been increased to 0',warning=.true.)
                tol_norm = 0._dp
            end if
            if (tol_norm.gt.1._dp) then
                call writo('tol_norm has been decreased to 1',warning=.true.)
                tol_norm = 1._dp
            end if

            ! check tol_rich
            if (tol_rich.lt.min_tol) then
                call writo('tol_rich has been increased to '//&
                    &trim(r2str(min_tol)),warning=.true.)
                tol_rich = min_tol
            end if
            if (tol_rich.gt.max_tol) then
                call writo('tol_rich has been decreased to '//&
                    &trim(r2str(max_tol)),warning=.true.)
                tol_rich = max_tol
            end if

            ! check tol_zero
            if (tol_zero.lt.min_tol) then
                call writo('tol_zero has been increased to '//&
                    &trim(r2str(min_tol)),warning=.true.)
                tol_zero = min_tol
            end if
            if (tol_zero.gt.max_tol) then
                call writo('tol_zero has been decreased to '//&
                    &trim(r2str(max_tol)),warning=.true.)
                tol_zero = max_tol
            end if

            ! check and setup tol_SLEPC
            allocate(tol_slepc_loc(max_it_rich))
            ! get maximum index set by user and correct
            max_id = 0
            do id = 1,max_size_tol_slepc
                if (tol_slepc(id).lt.huge(1._dp)) then                      ! was overwritten by user
                    max_id = id
                    if (tol_slepc(id).lt.min_tol) then
                        call writo('tol_SLEPC('//trim(i2str(id))//&
                            &') has been increased to '//trim(r2str(min_tol)),&
                            &warning=.true.)
                        tol_slepc(id) = min_tol
                    end if
                    if (tol_slepc(id).gt.max_tol) then
                        call writo('tol_SLEPC('//trim(i2str(id))//&
                            &') has been decreased to '//trim(r2str(max_tol)),&
                            &warning=.true.)
                        tol_slepc(id) = max_tol
                    end if
                end if
            end do
            ! set local variable
            if (max_id.gt.0) then                                           ! was overwritten by user
                if (max_id.gt.max_it_rich) then                             ! too many values given
                    call writo(trim(i2str(max_id-max_it_rich))//&
                        &' last values discarded for tol_SLEPC',warning=.true.)
                    tol_slepc_loc = tol_slepc(1:max_it_rich)
                else if (max_id.eq.max_it_rich) then                        ! exactly the right amount of values given
                    tol_slepc_loc = tol_slepc(1:max_it_rich)
                else                                                        ! not enough values given
                    call writo('tol_SLEPC('//trim(i2str(max_id+1))//':'//&
                        &trim(i2str(max_it_rich))//') has been set to '//&
```

```
                        &trim(r2str(tol_slepc_def))//' [default]',&
                        &warning=.true.)
                    tol_slepc_loc(1:max_id) = tol_slepc(1:max_id)
                    tol_slepc_loc(max_id+1:max_it_rich) = tol_slepc_def
                end if
            else                                                        ! was not overwritten by user
                tol_slepc_loc = tol_slepc_def                           ! all tolerances equal to default
            end if
        end subroutine adapt_tols

        ! Checks whether normalization variables are chosen correctly:
        !   rho_0 has to be positive.
        !> \private
        integer function adapt_normalization() result(ierr)
            character(*), parameter :: rout_name = 'adapt_normalization'

            ! local variables
            character(len=max_str_ln) :: err_msg                        ! error message

            ! initialize ierr
            ierr = 0

            if (rho_0.le.0) then
                ierr = 1
                err_msg = 'rho_0 has to be positive'
                chckerr(err_msg)
            end if
        end function adapt_normalization
    end function read_input_opts

    !> Reads the equilibrium input file if no Richardson restart.
    !!
    !! These variables will be passed on through the HDF5 data system.
    !!
    !! \see reconstruct_pb3d_in().
    !!
    !! \return ierr
    integer function read_input_eq() result(ierr)
        use num_vars, only: eq_style, use_pol_flux_e, eq_i
        use vmec_ops, only: read_vmec
        use helena_ops, only: read_hel
        use grid_vars, only: n_r_in

        character(*), parameter :: rout_name = 'read_input_eq'


        ! initialize ierr
        ierr = 0

        ! choose which equilibrium style is being used:
        !   1:  VMEC
        !   2:  HELENA
        select case (eq_style)
            case (1)                                                    ! VMEC
                ierr = read_vmec(n_r_in,use_pol_flux_e)
                chckerr('')
            case (2)                                                    ! HELENA
                ierr = read_hel(n_r_in,use_pol_flux_e)
                chckerr('')
        end select

        ! close equilibrium file
        close(eq_i)
    end function read_input_eq

    !> Print input quantities to an output file.
    !!
    !! Variables printed:
    !!  -  \c misc_in:  \c  prog_version,  \c eq_style,  \c  rho_style, \c  R_0,
    !!  \c  pres_0,   \c  B_0,  \c  psi_0   \c  rho_0,  \c  T_0,   \c  vac_perm,
    !!  \c   use_pol_flux_F,  \c   use_pol_flux_E,   \c  use_normalization,   \c
    !!  norm_disc_prec_eq, \c norm_disc_style_sol, \c n_r_in, \c n_r_eq, \c
    !!  n_r_sol, \c debug_version
    !!  -  \c misc_in_V: \c is_asym_V,  \c is_freeb_V, \c mnmax_V,  \c mpol_V, \c
    !!  ntor_V, \c gam_V
    !!  - \c flux_t_V
    !!  - \c flux_p_V
    !!  - \c pres_V
    !!  - \c rot_t_V
    !!  - \c q_saf_V
    !!  - \c mn_V
    !!  - \c R_V_c
    !!  - \c R_V_s
    !!  - \c Z_V_c
    !!  - \c Z_V_s
    !!  - \c L_V_c
    !!  - \c L_V_s
```

```
    !!  - \c B_V_sub: \c B_V_sub_c, \c B_V_sub_s
    !!  - \c B_V: \c B_V_c, \c B_V_s
    !!  - \c J_V_sup_int
    !!  - \c jac_V: \c jac_V_c, \c jac_V_s
    !!  - \c misc_in_H: \c ias, \c nchi
    !!  - \c RZ_H: \c R_H, \c Z_H
    !!  - \c chi_H
    !!  - \c q_saf_H
    !!  - \c rot_t_H
    !!  - \c RBphi_H
    !!  - \c pres_H
    !!  - \c flux_p_H
    !!  - \c flux_t_H
    !!  -  \c  misc_X:  \c  prim_X,  \c n_mod_X,  \c  min_sec_X,  \c  max_sec_X,
    !!  \c  norm_disc_prec_X,   \c  norm_style,  \c  U_style,   \c  X_style,  \c
    !!  matrix_SLEPC_style, \c K_style, \c alpha_style
    !!  - \c  misc_sol: \c  min_r_sol, \c  max_r_sol, \c  norm_disc_prec_sol, \c
    !!  BC_style, \c EV_BC, \c EV_BC
    !!
    !! \return ierr
    integer function print_output_in(data_name,remove_previous_arrs) &
        &result(ierr)

        use num_vars, only: eq_style, rho_style, prog_version, use_pol_flux_e, &
            &use_pol_flux_f, use_normalization, norm_disc_prec_eq, pb3d_name, &
            &norm_disc_prec_x, norm_style, u_style, x_style, alpha_style, &
            &matrix_slepc_style, bc_style, ev_style, norm_disc_prec_sol, &
            &ev_bc, magn_int_style, k_style, debug_version, plot_vmec_modes, &
            &norm_disc_style_sol, x_grid_style, max_njq_change
        use eq_vars, only: r_0, pres_0, b_0, psi_0, rho_0, t_0, vac_perm, &
            &max_flux_e, max_flux_f
        use grid_vars, only: n_r_in, n_r_eq, n_r_sol, n_alpha, min_alpha, &
            &max_alpha
        use grid_ops, only: calc_norm_range
        use x_vars, only: min_r_sol, max_r_sol, min_sec_x, max_sec_x, prim_x, &
            &n_mod_x
        use hdf5_ops, only: print_hdf5_arrs
        use hdf5_vars, only: dealloc_var_1d, var_1d_type, &
            &max_dim_var_1d
        use helena_vars, only: chi_h, flux_p_h, flux_t_h, r_h, z_h, nchi, ias, &
            &q_saf_h, rot_t_h, pres_h, rbphi_h
        use vmec_vars, only: is_freeb_v, mnmax_v, mpol_v, ntor_v, is_asym_v, &
            &gam_v, r_v_c, r_v_s, z_v_c, z_v_s, l_v_c, l_v_s, jac_v_c, &
            &jac_v_s, mnmax_v, mn_v, rot_t_v, q_saf_v, pres_v, flux_t_v, &
            &flux_p_v, nfp_v, b_v_sub_c, b_v_sub_s
        use helena_vars, only: h_h_11, h_h_12, h_h_33
#if ldebug
        use vmec_vars, only: b_v_c, b_v_s, j_v_sup_int
#endif

        character(*), parameter :: rout_name = 'print_output_in'

        ! input / output
        character(len=*), intent(in) :: data_name                              !< name under which to store
        logical, intent(in), optional :: remove_previous_arrs                  !< remove previous variables if present

        ! local variables
        type(var_1d_type), allocatable, target :: in_1d(:)                     ! 1D equivalent of input variables
        type(var_1d_type), pointer :: in_1d_loc => null()                      ! local element in in_1D
        integer :: id                                                          ! counter
        integer :: in_limits(2)                                                ! min. and max. index of input variable grid of this process
        integer :: loc_size                                                    ! local size

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Write input variables to output file')
        call lvl_ud(1)

        ! plot modes if requested and VMEC
        if (plot_vmec_modes) then
            call writo('Plotting decay of VMEC modes')
            call print_ex_2d('R_V_c','R_V_c',&
                &log10(maxval(abs(r_v_c(:,:,0)),2)),draw=.false.)
            call draw_ex(['R_V_c'],'R_V_c',1,1,.false.,ex_plot_style=1)
            call print_ex_2d('R_V_s','R_V_s',&
                &log10(maxval(abs(r_v_s(:,:,0)),2)),draw=.false.)
            call draw_ex(['R_V_s'],'R_V_s',1,1,.false.,ex_plot_style=1)
            call print_ex_2d('Z_V_c','Z_V_c',&
                &log10(maxval(abs(z_v_c(:,:,0)),2)),draw=.false.)
            call draw_ex(['Z_V_c'],'Z_V_c',1,1,.false.,ex_plot_style=1)
            call print_ex_2d('Z_V_s','Z_V_s',&
                &log10(maxval(abs(z_v_s(:,:,0)),2)),draw=.false.)
            call draw_ex(['Z_V_s'],'Z_V_s',1,1,.false.,ex_plot_style=1)
            call print_ex_2d('L_V_c','L_V_c',&
```

```
                &log10(maxval(abs(l_v_c(:,:,0)),2)),draw=.false.)
            call draw_ex(['L_V_c'],'L_V_c',1,1,.false.,ex_plot_style=1)
            call print_ex_2d('L_V_s','L_V_s',&
                &log10(maxval(abs(l_v_s(:,:,0)),2)),draw=.false.)
            call draw_ex(['L_V_s'],'L_V_s',1,1,.false.,ex_plot_style=1)
            call print_ex_2d('jac_V_c','jac_V_c',&
                &log10(maxval(abs(jac_v_c(:,:,0)),2)),draw=.false.)
            call draw_ex(['jac_V_c'],'jac_V_c',1,1,.false.,ex_plot_style=1)
            call print_ex_2d('jac_V_s','jac_V_s',&
                &log10(maxval(abs(jac_v_s(:,:,0)),2)),draw=.false.)
            call draw_ex(['jac_V_s'],'jac_V_s',1,1,.false.,ex_plot_style=1)
#if ldebug
            call print_ex_2d('B_V_c','B_V_c',&
                &log10(maxval(abs(b_v_c),2)),draw=.false.)
            call draw_ex(['B_V_c'],'B_V_c',1,1,.false.,ex_plot_style=1)
            call print_ex_2d('B_V_s','B_V_s',&
                &log10(maxval(abs(b_v_s),2)),draw=.false.)
            call draw_ex(['B_V_s'],'B_V_s',1,1,.false.,ex_plot_style=1)
#endif
        end if

        ! calculate limits of input range
        ierr = calc_norm_range('PB3D_in',in_limits=in_limits)
        chckerr('')
        n_r_eq = in_limits(2)-in_limits(1)+1
        if (in_limits(1).gt.1 .or. in_limits(2).lt.n_r_in) then
            call writo('Only the normal range '//trim(i2str(in_limits(1)))//&
                &'..'//trim(i2str(in_limits(2)))//' is written')
        else
            call writo('The entire normal range '//trim(i2str(in_limits(1)))//&
                &'..'//trim(i2str(in_limits(2)))//' is written')
        end if
        call writo('(relevant to solution range '//&
            &trim(r2strt(min_r_sol))//'..'//trim(r2strt(max_r_sol))//')')

        ! set up 1D equivalents of input variables
        allocate(in_1d(max_dim_var_1d))

        ! Set up common variables in_1D
        id = 1

        ! misc_in
        in_1d_loc => in_1d(id); id = id+1
        in_1d_loc%var_name = 'misc_in'
        allocate(in_1d_loc%tot_i_min(1),in_1d_loc%tot_i_max(1))
        allocate(in_1d_loc%loc_i_min(1),in_1d_loc%loc_i_max(1))
        in_1d_loc%loc_i_min = [1]
        in_1d_loc%loc_i_max = [20]
        in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
        in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
        allocate(in_1d_loc%p(20))
        in_1d_loc%p = [prog_version,eq_style*1._dp,rho_style*1._dp,&
            &r_0,pres_0,b_0,psi_0,rho_0,t_0,vac_perm,-1._dp,-1._dp,&
            &-1._dp,norm_disc_prec_eq*1._dp,n_r_in*1._dp,n_r_eq*1._dp,&
            &n_r_sol*1._dp,max_flux_e,max_flux_f,-1._dp]
        if (use_pol_flux_e) in_1d_loc%p(11) = 1._dp
        if (use_pol_flux_f) in_1d_loc%p(12) = 1._dp
        if (use_normalization) in_1d_loc%p(13) = 1._dp
        if (debug_version) in_1d_loc%p(20) = 1._dp

        ! misc_in_V or misc_in_H, depending on equilibrium style
        select case (eq_style)
            case (1)                                                    ! VMEC
                ! misc_in_V
                in_1d_loc => in_1d(id); id = id+1
                in_1d_loc%var_name = 'misc_in_V'
                allocate(in_1d_loc%tot_i_min(1),&
                    &in_1d_loc%tot_i_max(1))
                allocate(in_1d_loc%loc_i_min(1),&
                    &in_1d_loc%loc_i_max(1))
                in_1d_loc%loc_i_min = [1]
                in_1d_loc%loc_i_max = [7]
                in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
                in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
                allocate(in_1d_loc%p(7))
                in_1d_loc%p = [-1._dp,-1._dp,mnmax_v*1._dp,mpol_v*1._dp,&
                    &ntor_v*1._dp,nfp_v*1._dp,gam_v]
                if (is_asym_v) in_1d_loc%p(1) = 1._dp
                if (is_freeb_v) in_1d_loc%p(2) = 1._dp

                loc_size = n_r_eq*size(flux_t_v,2)

                ! flux_t_V
                in_1d_loc => in_1d(id); id = id+1
                in_1d_loc%var_name = 'flux_t_V'
                allocate(in_1d_loc%tot_i_min(2),in_1d_loc%tot_i_max(2))
                allocate(in_1d_loc%loc_i_min(2),in_1d_loc%loc_i_max(2))
```

```
                in_1d_loc%loc_i_min = [1,0]
                in_1d_loc%loc_i_max = [n_r_eq,size(flux_t_v,2)-1]
                in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
                in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
                allocate(in_1d_loc%p(loc_size))
                in_1d_loc%p = reshape(flux_t_v(in_limits(1):in_limits(2),:),&
                    &[loc_size])

                ! flux_p_V
                in_1d_loc => in_1d(id); id = id+1
                in_1d_loc%var_name = 'flux_p_V'
                allocate(in_1d_loc%tot_i_min(2),in_1d_loc%tot_i_max(2))
                allocate(in_1d_loc%loc_i_min(2),in_1d_loc%loc_i_max(2))
                in_1d_loc%loc_i_min = [1,0]
                in_1d_loc%loc_i_max = [n_r_eq,size(flux_p_v,2)-1]
                in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
                in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
                allocate(in_1d_loc%p(loc_size))
                in_1d_loc%p = reshape(flux_p_v(in_limits(1):in_limits(2),:),&
                    &[loc_size])

                loc_size = n_r_eq*size(pres_v,2)

                ! pres_V
                in_1d_loc => in_1d(id); id = id+1
                in_1d_loc%var_name = 'pres_V'
                allocate(in_1d_loc%tot_i_min(2),in_1d_loc%tot_i_max(2))
                allocate(in_1d_loc%loc_i_min(2),in_1d_loc%loc_i_max(2))
                in_1d_loc%loc_i_min = [1,0]
                in_1d_loc%loc_i_max = [n_r_eq,size(pres_v,2)-1]
                in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
                in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
                allocate(in_1d_loc%p(loc_size))
                in_1d_loc%p = reshape(pres_v(in_limits(1):in_limits(2),:),&
                    &[loc_size])

                ! rot_t_V
                in_1d_loc => in_1d(id); id = id+1
                in_1d_loc%var_name = 'rot_t_V'
                allocate(in_1d_loc%tot_i_min(2),in_1d_loc%tot_i_max(2))
                allocate(in_1d_loc%loc_i_min(2),in_1d_loc%loc_i_max(2))
                in_1d_loc%loc_i_min = [1,0]
                in_1d_loc%loc_i_max = [n_r_eq,size(rot_t_v,2)-1]
                in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
                in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
                allocate(in_1d_loc%p(loc_size))
                in_1d_loc%p = reshape(rot_t_v(in_limits(1):in_limits(2),:),&
                    &[loc_size])

                ! q_saf_V
                in_1d_loc => in_1d(id); id = id+1
                in_1d_loc%var_name = 'q_saf_V'
                allocate(in_1d_loc%tot_i_min(2),in_1d_loc%tot_i_max(2))
                allocate(in_1d_loc%loc_i_min(2),in_1d_loc%loc_i_max(2))
                in_1d_loc%loc_i_min = [1,0]
                in_1d_loc%loc_i_max = [n_r_eq,size(q_saf_v,2)-1]
                in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
                in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
                allocate(in_1d_loc%p(loc_size))
                in_1d_loc%p = reshape(q_saf_v(in_limits(1):in_limits(2),:),&
                    &[loc_size])

                ! mn_V
                in_1d_loc => in_1d(id); id = id+1
                in_1d_loc%var_name = 'mn_V'
                allocate(in_1d_loc%tot_i_min(2),in_1d_loc%tot_i_max(2))
                allocate(in_1d_loc%loc_i_min(2),in_1d_loc%loc_i_max(2))
                in_1d_loc%loc_i_min = [1,1]
                in_1d_loc%loc_i_max = [mnmax_v,2]
                in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
                in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
                allocate(in_1d_loc%p(size(mn_v)))
                in_1d_loc%p = reshape(mn_v,[size(mn_v)])

                ! RZL_V
                in_1d_loc => in_1d(id); id = id+1
                in_1d_loc%var_name = 'RZL_V'
                allocate(in_1d_loc%tot_i_min(4),in_1d_loc%tot_i_max(4))
                allocate(in_1d_loc%loc_i_min(4),in_1d_loc%loc_i_max(4))
                in_1d_loc%loc_i_min = [1,1,0,1]
                in_1d_loc%loc_i_max = [mnmax_v,n_r_eq,size(r_v_c,3)-1,6]
                in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
                in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
                allocate(in_1d_loc%p(6*mnmax_v*n_r_eq*size(r_v_c,3)))
                in_1d_loc%p = reshape([r_v_c(:,in_limits(1):in_limits(2),:),&
                    &r_v_s(:,in_limits(1):in_limits(2),:),&
                    &z_v_c(:,in_limits(1):in_limits(2),:),&
```

```
                    &z_v_s(:,in_limits(1):in_limits(2),:),&
                    &l_v_c(:,in_limits(1):in_limits(2),:),&
                    &l_v_s(:,in_limits(1):in_limits(2),:)],&
                    &[6*mnmax_v*n_r_eq*size(r_v_c,3)])

                ! jac_V
                in_1d_loc => in_1d(id); id = id+1
                in_1d_loc%var_name = 'jac_V'
                allocate(in_1d_loc%tot_i_min(4),in_1d_loc%tot_i_max(4))
                allocate(in_1d_loc%loc_i_min(4),in_1d_loc%loc_i_max(4))
                in_1d_loc%loc_i_min = [1,1,0,1]
                in_1d_loc%loc_i_max = [mnmax_v,n_r_eq,size(jac_v_c,3)-1,2]
                in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
                in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
                allocate(in_1d_loc%p(2*mnmax_v*n_r_eq*size(jac_v_c,3)))
                in_1d_loc%p = reshape([jac_v_c(:,in_limits(1):in_limits(2),:),&
                    &jac_v_s(:,in_limits(1):in_limits(2),:)],&
                    &[2*mnmax_v*n_r_eq*size(jac_v_c,3)])

                ! B_V_sub
                in_1d_loc => in_1d(id); id = id+1
                in_1d_loc%var_name = 'B_V_sub'
                allocate(in_1d_loc%tot_i_min(4),in_1d_loc%tot_i_max(4))
                allocate(in_1d_loc%loc_i_min(4),in_1d_loc%loc_i_max(4))
                in_1d_loc%loc_i_min = [1,1,1,1]
                in_1d_loc%loc_i_max = [mnmax_v,n_r_eq,3,2]
                in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
                in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
                allocate(in_1d_loc%p(2*mnmax_v*n_r_eq*3))
                in_1d_loc%p = reshape([&
                    &b_v_sub_c(:,in_limits(1):in_limits(2),:),&
                    &b_v_sub_s(:,in_limits(1):in_limits(2),:)],&
                    &[2*mnmax_v*n_r_eq*3])

#if ldebug
                ! B_V
                in_1d_loc => in_1d(id); id = id+1
                in_1d_loc%var_name = 'B_V'
                allocate(in_1d_loc%tot_i_min(3),in_1d_loc%tot_i_max(3))
                allocate(in_1d_loc%loc_i_min(3),in_1d_loc%loc_i_max(3))
                in_1d_loc%loc_i_min = [1,1,1]
                in_1d_loc%loc_i_max = [mnmax_v,n_r_eq,2]
                in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
                in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
                allocate(in_1d_loc%p(2*mnmax_v*n_r_eq))
                in_1d_loc%p = reshape([b_v_c(:,in_limits(1):in_limits(2)),&
                    &b_v_s(:,in_limits(1):in_limits(2))],[2*mnmax_v*n_r_eq])

                ! J_V_sup_int
                in_1d_loc => in_1d(id); id = id+1
                in_1d_loc%var_name = 'J_V_sup_int'
                allocate(in_1d_loc%tot_i_min(2),in_1d_loc%tot_i_max(2))
                allocate(in_1d_loc%loc_i_min(2),in_1d_loc%loc_i_max(2))
                in_1d_loc%loc_i_min = [1,1]
                in_1d_loc%loc_i_max = [n_r_eq,2]
                in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
                in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
                allocate(in_1d_loc%p(2*n_r_eq))
                in_1d_loc%p = reshape(j_v_sup_int(in_limits(1):in_limits(2),:),&
                    &[2*n_r_eq])

#endif
            case (2)                                                    ! HELENA
                ! misc_in_H
                in_1d_loc => in_1d(id); id = id+1
                in_1d_loc%var_name = 'misc_in_H'
                allocate(in_1d_loc%tot_i_min(1),&
                    &in_1d_loc%tot_i_max(1))
                allocate(in_1d_loc%loc_i_min(1),&
                    &in_1d_loc%loc_i_max(1))
                in_1d_loc%loc_i_min = [1]
                in_1d_loc%loc_i_max = [2]
                in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
                in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
                allocate(in_1d_loc%p(2))
                in_1d_loc%p = [ias*1._dp,nchi*1._dp]

                ! RZ_H
                in_1d_loc => in_1d(id); id = id+1
                in_1d_loc%var_name = 'RZ_H'
                allocate(in_1d_loc%tot_i_min(3),in_1d_loc%tot_i_max(3))
                allocate(in_1d_loc%loc_i_min(3),in_1d_loc%loc_i_max(3))
                in_1d_loc%loc_i_min = [1,1,1]
                in_1d_loc%loc_i_max = [nchi,n_r_eq,2]
                in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
                in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
                allocate(in_1d_loc%p(2*nchi*n_r_eq))
```

```
                in_1d_loc%p = reshape([r_h(:,in_limits(1):in_limits(2)),&
                    &z_h(:,in_limits(1):in_limits(2))],[2*nchi*n_r_eq])

                ! chi_H
                in_1d_loc => in_1d(id); id = id+1
                in_1d_loc%var_name = 'chi_H'
                allocate(in_1d_loc%tot_i_min(1),in_1d_loc%tot_i_max(1))
                allocate(in_1d_loc%loc_i_min(1),in_1d_loc%loc_i_max(1))
                in_1d_loc%loc_i_min = [1]
                in_1d_loc%loc_i_max = [nchi]
                in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
                in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
                allocate(in_1d_loc%p(nchi))
                in_1d_loc%p = chi_h

                loc_size = n_r_eq*size(flux_p_h,2)

                ! flux_p_H
                in_1d_loc => in_1d(id); id = id+1
                in_1d_loc%var_name = 'flux_p_H'
                allocate(in_1d_loc%tot_i_min(2),in_1d_loc%tot_i_max(2))
                allocate(in_1d_loc%loc_i_min(2),in_1d_loc%loc_i_max(2))
                in_1d_loc%loc_i_min = [1,0]
                in_1d_loc%loc_i_max = [n_r_eq,size(flux_p_h,2)-1]
                in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
                in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
                allocate(in_1d_loc%p(loc_size))
                in_1d_loc%p = reshape(flux_p_h(in_limits(1):in_limits(2),:),&
                    &[loc_size])

                ! flux_t_H
                in_1d_loc => in_1d(id); id = id+1
                in_1d_loc%var_name = 'flux_t_H'
                allocate(in_1d_loc%tot_i_min(2),in_1d_loc%tot_i_max(2))
                allocate(in_1d_loc%loc_i_min(2),in_1d_loc%loc_i_max(2))
                in_1d_loc%loc_i_min = [1,0]
                in_1d_loc%loc_i_max = [n_r_eq,size(flux_t_h,2)-1]
                in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
                in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
                allocate(in_1d_loc%p(loc_size))
                in_1d_loc%p = reshape(flux_t_h(in_limits(1):in_limits(2),:),&
                    &[loc_size])

                ! q_saf_H
                in_1d_loc => in_1d(id); id = id+1
                in_1d_loc%var_name = 'q_saf_H'
                allocate(in_1d_loc%tot_i_min(2),in_1d_loc%tot_i_max(2))
                allocate(in_1d_loc%loc_i_min(2),in_1d_loc%loc_i_max(2))
                in_1d_loc%loc_i_min = [1,0]
                in_1d_loc%loc_i_max = [n_r_eq,size(q_saf_h,2)-1]
                in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
                in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
                allocate(in_1d_loc%p(loc_size))
                in_1d_loc%p = reshape(q_saf_h(in_limits(1):in_limits(2),:),&
                    &[loc_size])

                ! rot_t_H
                in_1d_loc => in_1d(id); id = id+1
                in_1d_loc%var_name = 'rot_t_H'
                allocate(in_1d_loc%tot_i_min(2),in_1d_loc%tot_i_max(2))
                allocate(in_1d_loc%loc_i_min(2),in_1d_loc%loc_i_max(2))
                in_1d_loc%loc_i_min = [1,0]
                in_1d_loc%loc_i_max = [n_r_eq,size(rot_t_h,2)-1]
                in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
                in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
                allocate(in_1d_loc%p(loc_size))
                in_1d_loc%p = reshape(rot_t_h(in_limits(1):in_limits(2),:),&
                    &[loc_size])

                ! pres_H
                in_1d_loc => in_1d(id); id = id+1
                in_1d_loc%var_name = 'pres_H'
                allocate(in_1d_loc%tot_i_min(2),in_1d_loc%tot_i_max(2))
                allocate(in_1d_loc%loc_i_min(2),in_1d_loc%loc_i_max(2))
                in_1d_loc%loc_i_min = [1,0]
                in_1d_loc%loc_i_max = [n_r_eq,size(pres_h,2)-1]
                in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
                in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
                allocate(in_1d_loc%p(loc_size))
                in_1d_loc%p = reshape(pres_h(in_limits(1):in_limits(2),:),&
                    &[loc_size])

                ! RBphi_H
                in_1d_loc => in_1d(id); id = id+1
                in_1d_loc%var_name = 'RBphi_H'
                allocate(in_1d_loc%tot_i_min(2),in_1d_loc%tot_i_max(2))
                allocate(in_1d_loc%loc_i_min(2),in_1d_loc%loc_i_max(2))
```

```
                in_1d_loc%loc_i_min = [1,0]
                in_1d_loc%loc_i_max = [n_r_eq,size(rbphi_h,2)-1]
                in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
                in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
                allocate(in_1d_loc%p(loc_size))
                in_1d_loc%p = reshape(rbphi_h(in_limits(1):in_limits(2),:),&
                    &[loc_size])

                ! h_H
                in_1d_loc => in_1d(id); id = id+1
                in_1d_loc%var_name = 'h_H'
                allocate(in_1d_loc%tot_i_min(3),in_1d_loc%tot_i_max(3))
                allocate(in_1d_loc%loc_i_min(3),in_1d_loc%loc_i_max(3))
                in_1d_loc%loc_i_min = [1,1,1]
                in_1d_loc%loc_i_max = [nchi,n_r_eq,3]
                in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
                in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
                allocate(in_1d_loc%p(3*nchi*n_r_eq))
                in_1d_loc%p = reshape([h_h_11(:,in_limits(1):in_limits(2)),&
                    &h_h_12(:,in_limits(1):in_limits(2)),&
                    &h_h_33(:,in_limits(1):in_limits(2))],&
                    &[3*nchi*n_r_eq])
        end select

        ! misc_X
        in_1d_loc => in_1d(id); id = id+1
        in_1d_loc%var_name = 'misc_X'
        allocate(in_1d_loc%tot_i_min(1),in_1d_loc%tot_i_max(1))
        allocate(in_1d_loc%loc_i_min(1),in_1d_loc%loc_i_max(1))
        in_1d_loc%loc_i_min = [1]
        in_1d_loc%loc_i_max = [17]
        in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
        in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
        allocate(in_1d_loc%p(17))
        in_1d_loc%p = [prim_x*1._dp,n_mod_x*1._dp,min_sec_x*1._dp,&
            &max_sec_x*1._dp,norm_disc_prec_x*1._dp,norm_style*1._dp,&
            &u_style*1._dp,x_style*1._dp,matrix_slepc_style*1._dp,&
            &magn_int_style*1._dp,k_style*1._dp,alpha_style*1._dp,&
            &x_grid_style*1._dp,min_alpha,max_alpha,n_alpha*1._dp,&
            &max_njq_change]

        ! misc_sol
        in_1d_loc => in_1d(id); id = id+1
        in_1d_loc%var_name = 'misc_sol'
        allocate(in_1d_loc%tot_i_min(1),in_1d_loc%tot_i_max(1))
        allocate(in_1d_loc%loc_i_min(1),in_1d_loc%loc_i_max(1))
        in_1d_loc%loc_i_min = [1]
        in_1d_loc%loc_i_max = [8]
        in_1d_loc%tot_i_min = in_1d_loc%loc_i_min
        in_1d_loc%tot_i_max = in_1d_loc%loc_i_max
        allocate(in_1d_loc%p(8))
        in_1d_loc%p = [min_r_sol,max_r_sol,norm_disc_prec_sol*1._dp,&
            &norm_disc_style_sol*1._dp,bc_style(1)*1._dp,bc_style(2)*1._dp,&
            &ev_style*1._dp,ev_bc]

        ! write
        ierr = print_hdf5_arrs(in_1d(1:id-1),pb3d_name,trim(data_name),&
            &remove_previous_arrs=remove_previous_arrs)
        chckerr('')

        ! clean up
        call dealloc_var_1d(in_1d)
        nullify(in_1d_loc)

        ! user output
        call lvl_ud(-1)
    end function print_output_in
end module input_ops
```

D.46 /Users/toonweyens/Code/PB3D/Modules/input_utilities.f90 File Reference

Modules

- module input_utilities

  Numerical utilities related to input.

Functions/Subroutines

- logical function, public input_utilities::get_log (yes, ind)

  Queries for a logical value yes or no, where the default answer is also to be provided.
- real(dp) function, public input_utilities::get_real (lim_lo, lim_hi, ind)

  Queries for user input for a real value, where allowable range can be provided as well.
- integer function, public input_utilities::get_int (lim_lo, lim_hi, ind)

  Queries for user input for an integer value, where allowable range can be provided as well.
- subroutine, public input_utilities::pause_prog (ind)

  Pauses the running of the program.
- subroutine, public input_utilities::dealloc_in ()

  Cleans up input from equilibrium codes.

D.47 input_utilities.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------!
!> Numerical utilities related to input.
!-----------------------------------------------------------------------!
module input_utilities
#include <PB3D_macros.h>
    use str_utilities
    use output_ops
    use messages
    use num_vars, only: dp, max_str_ln

    implicit none
    private
    public pause_prog, get_real, get_int, get_log, dealloc_in

contains
    !> Queries for a logical  value yes or no, where the  default answer is also
    !! to be provided.
    !!
    !! If not called by all processes at the  same time, \c ind should be set to
    !! indicate an individual operation.
    function get_log(yes,ind) result(val)
        use num_vars, only: rank
        use mpi_utilities, only: broadcast_var

        ! input / output
        logical :: val                                                  !< output value
        logical :: yes                                                  !< default answer
        logical, intent(in), optional :: ind                            !< individual pause or not

        ! local variables
        character(len=11) :: empty_str = ''                             ! empty string
        character(len=max_str_ln) :: answer_str                         ! string with answer
        integer :: istat                                                ! status
        logical :: ind_loc                                              ! local version of ind

        ! set local ind
        ind_loc = .false.
        if (present(ind)) ind_loc = ind

        ! only master can receive input
        if (rank.eq.0) then
            write(*,'(A)',advance='no') empty_str                       ! first print empty string so that output is visible
            if (yes) then
                write(*,'(A)',advance='no') 'y(es)/n(o) [yes]: '
                val = .true.
            else
                write(*,'(A)',advance='no') 'y(es)/n(o) [no]: '
                val = .false.
            end if
            call stop_time
            read (*, '(A)') answer_str
            call start_time

            select case (strh2l(trim(answer_str)))
                !case ('y','yes')
```

```
                case ('y','Y','yes','Yes','YEs','YES','yEs','yES','yeS','YeS')
                    val = .true.
                case ('n','N','no','No','NO','nO')
                    val = .false.
            end select
        end if

        ! if not individual, broadcast result
        if (.not.ind_loc) then
            istat = broadcast_var(val)
            if (istat.ne.0) call writo('In get_log, something went wrong. &
                &Default used.',warning=.true.)
        end if
    end function get_log

    !> Queries for  user input for  a real value,  where allowable range  can be
    !! provided as well.
    !!
    !! If not called by all processes at the  same time, \c ind should be set to
    !! indicate an individual operation.
    function get_real(lim_lo,lim_hi,ind) result(val)
        use num_vars, only: rank
        use mpi_utilities, only: broadcast_var

        ! input / output
        real(dp) :: val                                                 !< output value
        real(dp), intent(in), optional :: lim_lo                        !< upper and lower limit
        real(dp), intent(in), optional :: lim_hi                        !< upper and lower limit
        logical, intent(in), optional :: ind                            !< individual pause or not

        ! local variables
        character(len=11) :: empty_str = ''                             ! empty string
        integer :: istat                                                ! status
        logical :: ind_loc                                              ! local version of ind
        real(dp) :: lims_loc(2)                                         ! local version of limits

        ! set local ind
        ind_loc = .false.
        if (present(ind)) ind_loc = ind

        ! initialize val
        val = 0._dp

        ! only master can receive input
        if (rank.eq.0) then
            ! set up local limits
            lims_loc = [-huge(1._dp),huge(1._dp)]
            if (present(lim_lo)) lims_loc(1) = lim_lo
            if (present(lim_hi)) lims_loc(2) = lim_hi

            ! get input
            do
                write(*,'(A)',advance='no') empty_str                   ! first print empty string so that output is visible
                write(*,'(A)',advance='no') 'Input a value'
                if (present(lim_lo).or.present(lim_hi)) then
                    write(*,'(A)',advance='no') ' ['
                    if (present(lim_lo)) write(*,'(A)',advance='no') &
                        &trim(r2strt(lim_lo))
                    write(*,'(A)',advance='no') '..'
                    if (present(lim_hi)) write(*,'(A)',advance='no') &
                        &trim(r2strt(lim_hi))
                    write(*,'(A)',advance='no') ']'
                end if
                write(*,'(A)',advance='no') ': '
                call stop_time
                read (*,*,iostat=istat) val
                call start_time

                if (istat.ne.0 .or. val.lt.lims_loc(1) .or. val.gt. &
                    &lims_loc(2)) then
                    write(*,'(A)',advance='no') empty_str               ! first print empty string so that output is visible
                    write(*,'(A)',advance='no') 'Choose a value between '//&
                        &trim(r2strt(lims_loc(1)))//' and '//&
                        &trim(r2strt(lims_loc(2)))
                    write(*,*) ''
                    cycle
                else
                    exit
                end if
            end do
        end if

        ! if not individual, broadcast result
        if (.not.ind_loc) then
            istat = broadcast_var(val)
```

```
            if (istat.ne.0) call writo('In get_real, something went &
                &wrong. Default of zero used.',warning=.true.)
        end if
    end function get_real

    !> Queries for user input for an integer value, where allowable range can be
    !! provided as well.
    !!
    !! If not called by all processes at the  same time, \c ind should be set to
    !! indicate an individual operation.
    function get_int(lim_lo,lim_hi,ind) result(val)
        use num_vars, only: rank
        use mpi_utilities, only: broadcast_var

        ! input / output
        integer :: val                                                  !< output value
        integer, intent(in), optional :: lim_lo                         !< upper and lower limit
        integer, intent(in), optional :: lim_hi                         !< upper and lower limit
        logical, intent(in), optional :: ind                            !< individual pause or not

        ! local variables
        character(len=11) :: empty_str = ''                             ! empty string
        integer :: istat                                                ! status
        logical :: ind_loc                                              ! local version of ind
        integer :: lims_loc(2)                                         ! local version of limits

        ! set local ind
        ind_loc = .false.
        if (present(ind)) ind_loc = ind

        ! initialize val
        val = 0

        ! only master can receive input
        if (rank.eq.0) then
            ! set up local limits
            lims_loc = [-huge(1),huge(1)]
            if (present(lim_lo)) lims_loc(1) = lim_lo
            if (present(lim_hi)) lims_loc(2) = lim_hi

            ! get input
            do
                write(*,'(A)',advance='no') empty_str                   ! first print empty string so that output is visible
                write(*,'(A)',advance='no') 'Input a value'
                if (present(lim_lo).or.present(lim_hi)) then
                    write(*,'(A)',advance='no') ' ['
                    if (present(lim_lo)) write(*,'(A)',advance='no') &
                        &trim(i2str(lim_lo))
                    write(*,'(A)',advance='no') '..'
                    if (present(lim_hi)) write(*,'(A)',advance='no') &
                        &trim(i2str(lim_hi))
                    write(*,'(A)',advance='no') ']'
                end if
                write(*,'(A)',advance='no') ': '
                call stop_time
                read (*,*,iostat=istat) val
                call start_time

                if (istat.ne.0 .or. val.lt.lims_loc(1) .or. val.gt. &
                    &lims_loc(2)) then
                    write(*,'(A)',advance='no') empty_str               ! first print empty string so that output is visible
                    write(*,'(A)',advance='no') 'Choose a value between '//&
                        &trim(i2str(lims_loc(1)))//' and '//&
                        &trim(i2str(lims_loc(2)))
                    write(*,*) ''
                    cycle
                else
                    exit
                end if
            end do
        end if

        ! if not individual, broadcast result
        if (.not.ind_loc) then
            istat = broadcast_var(val)
            if (istat.ne.0) call writo('In get_int, something went &
                &wrong. Default of zero used.',warning=.true.)
        end if
    end function get_int

    !> Pauses the running of the program.
    !!
    !! If not called by all processes at the  same time, \c ind should be set to
    !! indicate an individual operation.
    subroutine pause_prog(ind)
```

```
        use mpi_utilities, only: wait_mpi
        use num_vars, only: rank, rank

        ! input / output
        logical, intent(in), optional :: ind                                    ! individual pause or not

        ! local variables
        character(len=11) :: empty_str = ''                                     ! empty string
        integer :: istat                                                        ! status
        logical :: ind_loc                                                      ! local version of ind
        character(len=max_str_ln) :: hidden_msg                                 ! hidden message

        ! output message
        if (rank.eq.0) then
            write(*,'(A)',advance='no') empty_str                               ! first print empty string so that output is visible
            write(*,'(A)',advance='no') 'Paused. Press enter...'
        end if

        ! set local ind
        ind_loc = .false.
        if (present(ind)) ind_loc = ind

        ! only master can receive input
        if (rank.eq.0) then
            call stop_time
            read (*,'(A)') hidden_msg
            call start_time
        end if

        ! wait for MPI
        if (.not.ind_loc) then
            istat = wait_mpi()
            if (istat.ne.0) call writo('In pause_prog, something went &
                &wrong. Continuing.',warning=.true.)
        end if

        ! hidden message
        if (trim(hidden_msg).eq.'stop') stop 0
    end subroutine pause_prog

    !> Cleans up input from equilibrium codes.
    subroutine dealloc_in()
        use num_vars, only: eq_style
        use vmec_vars, only: dealloc_vmec
        use helena_vars, only: dealloc_hel

        ! deallocate depending on equilibrium style
        select case (eq_style)
            case (1)                                                            ! VMEC
                call dealloc_vmec
            case (2)                                                            ! HELENA
                call dealloc_hel
        end select
    end subroutine dealloc_in
end module input_utilities
```

D.48 /Users/toonweyens/Code/PB3D/Modules/messages.f90 File Reference

Modules

- module messages

  Numerical utilities related to giving output.

Functions/Subroutines

- subroutine, public messages::init_output ()

  Initialize the variables for the module.
- subroutine, public messages::print_hello ()

Prints first message.

- subroutine, public messages::print_goodbye ()

  Prints last messag.
- subroutine, public messages::init_time ()

  Intialize the time passed to 0.
- subroutine, public messages::start_time ()

  Start a timer.
- subroutine, public messages::stop_time ()

  Stop a timer.
- subroutine, public messages::passed_time ()

  Display the time that has passed between `t1` and `t2`.
- subroutine, public messages::print_err_msg (err_msg, routine_name)

  Prints an error message that is either user-provided, or the name of the calling routine.
- subroutine, public messages::lvl_ud (inc)

  Increases/decreases `lvl` of output.
- subroutine, public messages::writo (input_str, persistent, error, warning, alert)

  Write output to file identified by `output_i`.
- subroutine, public messages::print_ar_2 (arr)

  Print an array of dimension 2 on the screen.
- subroutine, public messages::print_ar_1 (arr)

  Print an array of dimension 1 on the screen.
- integer function, public messages::get_mem_usage ()

  Returns the memory usage in kilobytes.

Variables

- integer, public messages::lvl

  determines the indenting. higher `lvl` = more indenting
- character(len=2), public messages::lvl_sep = ''

  characters that separate different levels of output
- character(len=10), public messages::time_sep = ''

  defines the length of time part of output
- logical, public messages::temp_output_active

  true if temporary output is to be written in `temp_output`
- character(len=max_str_ln), dimension(:), allocatable, public messages::temp_output

  temporary output, before output file is opened

D.49 messages.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Numerical utilities related to giving output.
!-----------------------------------------------------------------------------!
module messages
    use str_utilities
    use num_vars, only: dp, max_str_ln
    use foul

    implicit none
    private
    public init_output, lvl_ud, writo, print_ar_1, print_ar_2, &
        &print_err_msg, init_time, start_time, stop_time, passed_time, &
        &print_hello, print_goodbye, &
```

```
        &temp_output, lvl, lvl_sep, temp_output_active, time_sep
#if ldebug
    public get_mem_usage
#endif

    ! global variables
    integer :: lvl                                                      !< determines the indenting. higher \c lvl = more indenting
    character(len=2) :: lvl_sep = ''                                    !< characters that separate different levels of output
    character(len=10) :: time_sep = ''                                  !< defines the length of time part of output
    real(dp) :: deltat                                                  !< length of time interval
    real(dp) :: t1, t2                                                  !< end points of time interval
    logical :: running                                                  !< whether the timer is running
    logical :: temp_output_active                                       !< true if temporary output is to be written in \c temp_output
    character(len=max_str_ln), allocatable :: temp_output(:)            !< temporary output, before output file is opened

contains
    !> Initialize the variables for the module.
    subroutine init_output()
        use num_vars, only: rank
#if ldebug
        use num_vars, only: mem_usage_count
#endif

        ! output level
        lvl = 1

        ! time
        deltat = 0
        t1 = 0
        t2 = 0
        running = .false.

        ! temporary output for master
        if (rank.eq.0) then
            temp_output_active = .true.
            allocate(temp_output(0))
        else
            temp_output_active = .false.
        end if

#if ldebug
        mem_usage_count = 0
#endif
    end subroutine init_output

    !> Prints first message.
    subroutine print_hello()
        use num_vars, only: rank, prog_name, prog_version, n_procs, &
            &debug_version

        ! local variables
        integer :: istat                                                ! status

        if (rank.eq.0) then
            call write_formatted(' PB3D Copyright (C) 2019  Toon Weyens',&
                &'italic')
            call write_formatted(' This program comes with ABSOLUTELY NO &
                &WARRANTY.','italic')
            call write_formatted(' This is free software, and you are welcome &
                &to redistribute it','italic')
            call write_formatted(' under certain conditions; See LICENSE for &
                &details.','italic')
            call write_formatted('','italic')

            call write_formatted(' Simulation started on '//get_date()//', at '&
                &//get_clock(),'italic')
            call write_formatted(' '//prog_name//' version: '//&
                &trim(r2strt(prog_version)),'italic')
            if (debug_version) then
                call write_formatted(' debug version','italic')
            else
                call write_formatted(' release version','italic')
            end if
            if (n_procs.eq.1) then
                call write_formatted(' 1 MPI process','italic')
            else
                call write_formatted(' '//trim(i2str(n_procs))//&
                    &' MPI processes','italic')
            end if
            write(*,*,iostat=istat) ''
        end if
    end subroutine print_hello

```

```fortran
    !> Prints last messag.
    subroutine print_goodbye()
        use num_vars, only: rank

        ! local variables
        integer :: istat                                                ! status

        if (rank.eq.0) then
            write(*,*,iostat=istat) ''
            call write_formatted(' Simulation finished on '//get_date()//&
                &', at '//get_clock(),'italic')
        end if
    end subroutine print_goodbye

    !> Intialize the time passed to 0.
    subroutine init_time()
        deltat = 0
        t1 = 0
        t2 = 0
        running = .false.
    end subroutine init_time

    !> Start a timer.
    subroutine start_time()
        if (running) then
            call writo('Tried to start timer, but was already running',&
                &warning=.true.)
        else
            call cpu_time(t1)
            running = .true.
        end if
    end subroutine start_time

    !> Stop a timer.
    subroutine stop_time()
        if (running) then
            call cpu_time(t2)

            ! increase deltat
            deltat = deltat+t2-t1

            ! set t1 and t2 back to zero
            t1 = 0
            t2 = 0
            running = .false.
        else
            call writo('Tried to stop timer, but was already stopped',&
                &warning=.true.)
        end if
    end subroutine stop_time

    !> Display the time that has passed between \c t1 and \c t2.
    !!
    !! Automatically stops time and resets everything to zero.
    subroutine passed_time()
        ! local variables
        character(len=max_str_ln) :: begin_str, end_str
        integer :: time_in_units(4)                                     ! time in days, hours, minutes, seconds
        integer :: total_time
        character(len=6) :: time_units(4) = ['day   ','hour  ','minute',&
            &'second']
        integer :: id                                                   ! counter

        ! stop at current time if running
        if (running) call stop_time

        begin_str = '(this took'
        if (deltat.lt.1) then
            end_str = ' less than 1 second)'
        else
            ! get days, hours, minutes, seconds
            total_time = floor(deltat)                                  ! get total time in seconds
            time_in_units(1) = total_time/(60*60*24)                    ! get days
            total_time = mod(total_time,60*60*24)                       ! subtract days from total time
            time_in_units(2) = total_time/(60*60)                       ! get hours
            total_time = mod(total_time,60*60)                          ! subtract hours from total time
            time_in_units(3) = total_time/(60)                          ! get minutes
            total_time = mod(total_time,60)                             ! subtract minutes from total time
            time_in_units(4) = total_time                               ! get seconds

            ! set up end_str
            end_str = ''
            do id = 1,4
                if (time_in_units(id).gt.0) then
                    if (trim(end_str).ne.'') end_str = trim(end_str)//','
                    end_str = trim(end_str)//' '//&
                        &trim(i2str(time_in_units(id)))//' '//&
```

```
                        &trim(time_units(id))
                    if (time_in_units(id).gt.1) end_str = trim(end_str)//'s'
                end if
            end do
            end_str = trim(end_str)//')'
        end if
        call writo(trim(begin_str) // trim(end_str))

        ! restart deltat
        call init_time
    end subroutine passed_time

    !> Returns the date.
    !!
    !! from <http://infohost.nmt.edu/tcc/help/lang/fortran/date.html>
    function get_date() result(now)
#if (lwith_intel && !lwith_gnu)
        use ifport
#endif
        ! input / output
        character(len=10) :: now                                        !< date

        ! local variables
        integer :: today(3)

        call idate(today)                                               ! today(1)=day, (2)=month, (3)=year

        write (now,'(i2.2,"/",i2.2,"/",i4.4)')  today(2), today(1), today(3)
    end function get_date

    !> Returns the time.
    !!
    !! from <http://infohost.nmt.edu/tcc/help/lang/fortran/date.html>
    function get_clock() result(time)
        ! input / output
        character(len=8) :: time                                        !< time

        ! local variables
        integer :: now(3)

        call itime(now)                                                 ! now(1)=hour, (2)=minute, (3)=second

        write (time,'(i2.2,":",i2.2,":",i2.2)')  now
    end function get_clock

    !> Prints an error message that is  either user-provided, or the name of the
    !! calling routine.
    !!
    !! \note This should be used with the macro CHCKERR.
    subroutine print_err_msg(err_msg,routine_name)
        use num_vars, only: rank

        ! input / output
        character(len=*), intent(in) :: err_msg                         !< error message to be printed
        character(len=*), intent(in) :: routine_name                    !< name of the routine

        if (trim(err_msg).eq.'') then
            lvl = 2
            call writo('>> calling routine: '//trim(routine_name)//' of rank '&
                &//trim(i2str(rank)),persistent=.true.)
        else
            call writo('ERROR in '//trim(routine_name)//': '//trim(err_msg),&
                &persistent=.true.,error=.true.)
        end if
    end subroutine print_err_msg

    !> Increases/decreases \c lvl of output.
    !!
    !! Name stands for level up or down.
    subroutine lvl_ud(inc)
        ! input / output
        integer :: inc                                                  !< increment of level

        if (lvl+inc.lt.1) then
            lvl = 1
            call writo('cannot go below lowest level',warning=.true.)
        else
            lvl = lvl + inc
        end if
    end subroutine lvl_ud

    !> Write output to file identified by \c output_i.
    !!
    !! This is done using the correct  indentation for the level (\c lvl_loc) of
    !! the output.
    !!
    !! By  default, only the  master outputs, but this  can be changed  using \c
```

```
    !! persistent.
    !!
    !! Optionally, special formatting for error, warning or alert can be chosen.
    subroutine writo(input_str,persistent,error,warning,alert)
        use num_vars, only: rank, output_i, no_output, max_tot_mem, max_x_mem
#if ldebug
        use mpi
        use num_vars, only: print_mem_usage, prog_name, mem_usage_name, &
            &mem_usage_i, mem_usage_count, time_start
#endif

        ! input / output
        character(len=*), intent(in) :: input_str                                !< the name that is searched for
        logical, intent(in), optional :: persistent                              !< output even if not group master
        logical, intent(in), optional :: error                                   !< error message
        logical, intent(in), optional :: warning                                 !< warning message
        logical, intent(in), optional :: alert                                   !< alert message

        ! local variables
        !character(len=7), parameter :: bright_str = "bright "                    ! dark terminal theme
        character(len=0), parameter :: bright_str = ""                           ! light terminal theme
        character(len=max_str_ln) :: output_str                                  ! output string
        character(len=max_str_ln) :: time_str                                    ! time string
        character(len=max_str_ln) :: header_str                                  ! header string
        character(len=max_str_ln) :: input_str_loc                               ! local input string
        character(len=max_str_ln), allocatable :: temp_output_loc(:)             ! local temporary output
        integer :: id, i_part                                                    ! counters
        integer :: max_len_part, num_parts, st_part, en_part                     ! variables controlling strings
        logical :: ignore                                                        ! normally, everybody but group master is ignored
        logical :: error_loc                                                     ! local error
        logical :: warning_loc                                                   ! local warning
        logical :: alert_loc                                                     ! local alert
        integer :: istat                                                         ! status
#if ldebug
        integer :: mem_usage                                                     ! memory usage
        integer(kind=8) :: clock                                                 ! current clock
#endif

        ! bypass output if no_output
        if (no_output) return

        ! setup ignore, error, warning and alert
        ignore = .true.                                                          ! ignore by default
        if (rank.eq.0) ignore = .false.                                          ! master process can output
        if (present(persistent)) ignore = .not.persistent                        ! persistent can override this
        error_loc = .false.
        if (present(error)) error_loc = error
        warning_loc = .false.
        if (present(warning)) warning_loc = warning
        alert_loc = .false.
        if (present(alert)) alert_loc = alert

        ! set local input string
        input_str_loc = input_str

        ! prepend "WARNING: " if warning
        if (warning_loc) input_str_loc = 'WARNING: '//trim(input_str_loc)

#if ldebug
        ! memory usage
        if (print_mem_usage) then
            ! increment counter
            mem_usage_count = mem_usage_count + 1

            ! get memory usage
            mem_usage = get_mem_usage()

            ! append count and memory usage in MegaBytes
            input_str_loc = trim(input_str_loc)//' - ['//&
                &trim(i2str(mem_usage_count))//': '//&
                &trim(i2str(mem_usage))//'kB]'

            ! get clock
            call system_clock(clock)

            ! write rank, count, time, memory usage to file if not temp_output
            if (.not.temp_output_active) then
                open(unit=mem_usage_i,file=prog_name//'_'//&
                    &trim(mem_usage_name)//'.dat',status='old',&
                    &position='append',iostat=istat)
                if (istat.eq.0) then
                    write(mem_usage_i,"(1X,2I10,I21,I10,2ES23.16)",&
                        &iostat=istat) &
                        &rank, mem_usage_count, clock-time_start, mem_usage, &
                        &max_tot_mem*1000, max_x_mem*1000
                    close(unit=mem_usage_i,iostat=istat)
```

```
                end if
            end if
        end if
#endif

        if (.not.ignore) then                                          ! only group master (= global master if no groups) or persistent
            ! set local error

            ! Divide the input string length by the max_str_ln and loop over the
            ! different parts
            max_len_part = max_str_ln-(lvl-1)*len(lvl_sep) - len(time_sep)      ! max length of a part including time part
            num_parts = (len(trim(input_str_loc))-1)/(max_len_part) + 1         ! how many parts there are
            do i_part = 1, num_parts
                ! construct input string for the appropriate level
                st_part = (i_part-1)*max_len_part+1                            ! index of start of this part
                if (i_part.lt.num_parts) then                                  ! index of end of this part
                    en_part = i_part*max_len_part
                else                                                           ! last part is shorter
                    en_part = len(trim(input_str_loc))
                end if
                output_str = input_str_loc(st_part:en_part)
                call format_str(lvl,output_str)
                call get_time_str(time_str)

                ! construct header string of equal length as output strength
                header_str = ''
                do id = 1, len(trim(output_str)) + len(trim(time_str)) + 1 - &
                    &len(time_sep)                                             ! not including time part, 1 for the space
                    header_str =  trim(header_str) // '-'
                end do
                header_str =  '          '//trim(header_str)                   ! number of spaces matches time string

                ! write output to file output_i or to temporary output
                if (temp_output_active) then                                   ! temporary output to internal variable temp_output
                    ! back up previous temporary output in local variable
                    allocate(temp_output_loc(size(temp_output)))
                    temp_output_loc = temp_output

                    ! reallocate temp_output
                    deallocate(temp_output)
                    if (lvl.eq.1) then                                         ! first level
                        allocate(temp_output(size(temp_output_loc)+3))
                        temp_output(1:size(temp_output_loc)) = temp_output_loc
                        temp_output(size(temp_output_loc)+1) = trim(header_str) ! first level gets extra lines
                        temp_output(size(temp_output_loc)+2) = trim(time_str)//&
                            &' '//trim(output_str)
                        temp_output(size(temp_output_loc)+3) = trim(header_str)
                    else                                                       ! other levels only need one line
                        allocate(temp_output(size(temp_output_loc)+1))
                        temp_output(1:size(temp_output_loc)) = temp_output_loc
                        temp_output(size(temp_output_loc)+1) = trim(time_str)//&
                            &' '//output_str
                    end if

                    ! deallocate local variable
                    deallocate(temp_output_loc)
                else                                                           ! normal output to file output_i
                    if (output_i.ne.0) then
                        if (lvl.eq.1) write(output_i,"(1X,A)",iostat=istat) &
                            &trim(header_str)                                  ! first level gets extra lines
                        write(output_i,"(1X,A)",iostat=istat) &
                            &trim(time_str)//' '//trim(output_str)
                        if (lvl.eq.1) write(output_i,"(1X,A)",iostat=istat) &
                            &trim(header_str)
                    end if
                end if

                ! also write output to screen
                if (error_loc) then
                    call write_formatted(' '//trim(time_str)//' ',&
                        &'background_red',trim(output_str),'italic underline')
                else if (warning_loc .or. alert_loc) then
                    call write_formatted(' '//trim(time_str)//' ',&
                        &'background_cyan',trim(output_str),'italic underline')
                else if (lvl.eq.1) then
                    call write_formatted(' '//trim(header_str),&
                        &bright_str//'green')
                    call write_formatted(' '//trim(time_str)//' ','',&
                        &trim(output_str),bright_str//'green')
                    call write_formatted(' '//trim(header_str),&
                        &bright_str//'green')
                else if (lvl.eq.2) then
                    call write_formatted(' '//trim(time_str)//' ','',&
                        &trim(output_str),bright_str//'blue')
                else
```

```
                    write(*,"(1X,A)",iostat=istat) &
                        &trim(time_str)//' '//trim(output_str)
                end if
            end do
        end if
    contains
        ! formats the string, merging time information with the string.
        !> \private
        subroutine format_str(lvl,str)
            ! input / output
            integer, intent(in) :: lvl                                  ! lvl of indentation
            character(len=*), intent(inout) :: str                      ! string that is to be formatted for the lvl

            ! local variables
            integer :: id                                               ! counter

            do id = 1,lvl-1                                             ! start with lvl 1
                str = lvl_sep // trim(str)
            end do
        end subroutine format_str

        ! prints a string with time information
        !> \private
        subroutine get_time_str(time_str)
            ! input / output
            character(len=*), intent(inout) :: time_str                 ! string with time information

            time_str = get_clock()//':'
        end subroutine get_time_str
    end subroutine writo

    !> Print an array of dimension 2 on the screen.
    subroutine print_ar_2(arr)
        ! input / output
        real(dp), intent(in) :: arr(:,:)                                !< array to be printed

        integer :: id

        do id=1,size(arr,1)
            call print_ar_1(arr(id,:))
        end do
    end subroutine print_ar_2

    !> Print an array of dimension 1 on the screen.
    subroutine print_ar_1(arr)
        ! input / output
        real(dp), intent(in) :: arr(:)                                  !< array to be printed

        ! local variables
        integer :: id
        character(len=2*max_str_ln) :: output_str                       ! holds string for a range of values
        character(len=14) :: var_str                                    ! holds string for one value, for last value
        integer :: vlen                                                 ! space for one variable + leading space
        integer :: n_free                                               ! how much space free
        logical :: str_full                                             ! whether the output_str is full
        integer :: istat                                                ! status

        output_str = '|'
        str_full = .false.
        id = 1
        vlen = 10                                                       ! 9 for variable and 1 for space

        do while(.not.str_full .and. id.le.size(arr))
            var_str = ''
            n_free = len(output_str) - len(trim(output_str))            ! how many characters free

            if (n_free.ge.2*vlen+6) then                                ! >2 fit
                continue
            else if (2*vlen+2.le.n_free .and. n_free.lt.2*vlen+6 .and. &  ! special case: maximum 2 left -> all fit
                &id+1.ge.size(arr)) then
                    continue
            else if (vlen+6.le.n_free .and. n_free.lt.2*vlen+6) then    ! 1 fit, rest truncated
                if (id.ge.size(arr)) then                               ! only 1 left -> all fit
                    continue
                else                                                    ! truncate
                    str_full = .true.
                end if
            else if (vlen+2.le.n_free .and. n_free.lt.vlen+6 .and. &    ! special case: maximum 1 left -> all fit
                &id.eq.size(arr)) then
                continue
            else                                                        ! this should never be reached
                call writo('not enough room to display variable. Something &
                    &went wrong',warning=.true.)
                return
            end if
```

```
            if (str_full) then                                          ! truncate
                write (var_str, '(ES9.2)') arr(size(arr))               ! last variable
                var_str = ' ... ' // trim(var_str)
            else
                write (var_str, '(ES9.2)') arr(id)                      ! current variable
                var_str = ' ' // trim(var_str)
            end if
            output_str = trim(output_str) // trim(var_str)

            id = id+1
        end do
        output_str = trim(output_str) // ' |'
        write(*,'(1X,A)',iostat=istat) output_str
    end subroutine print_ar_1

#if ldebug
    !> Returns the memory usage in kilobytes.
    !!
    !! Based on
    !! <http://stackoverflow.com/questions/22028571/track-memory-usage-in-fortran-90>
    !!
    !! \note Only works under linux.
    !!
    !! \ldebug
    integer function get_mem_usage() result(mem)
        use num_vars, only: mem_usage_i
#if ( lwith_intel && !lwith_gnu)
        use ifport
#endif

        ! local variables
        character(len=200):: filename=' '                               !< name of file where memory stored
        character(len=80) :: line                                       !< line of memory file
        character(len=8)  :: pid_char=' '                               !< process ID in string
        logical :: exists                                               !< whether memory file exists
        integer :: pid                                                  !< process ID
        integer :: istat                                                !< status

        ! initiazlie mem to negative number
        mem = -1

        ! get process ID and write it in string
        pid = getpid()
        write(pid_char,'(I8)') pid

        ! set up file name
        filename='/proc/'//trim(adjustl(pid_char))//'/status'

        ! inquire about memory file
        inquire (file=filename,exist=exists)
        if (.not.exists) return

        ! open and read memory file
        open(unit=mem_usage_i-1,file=filename,action='read',iostat=istat)
        if (istat.ne.0) return
        do
            read (mem_usage_i-1,'(A)',iostat=istat) line
            if (istat.ne.0) return
            if (line(1:6).eq.'VmRSS:') then
                read (line(7:),*) mem
                exit
            endif
        end do

        ! close memory file
        close(mem_usage_i-1)
    end function get_mem_usage
#endif
end module messages
```

D.50 /Users/toonweyens/Code/PB3D/Modules/MPI_ops.f90 File Reference

Modules

- module mpi_ops

  Operations related to MPI.

Functions/Subroutines

- integer function, public mpi_ops::start_mpi ()

  Start MPI and gather information.
- integer function, public mpi_ops::stop_mpi (grid_eq, grid_eq_b, grid_x, grid_x_b, grid_sol, eq_1, eq_2, x_1, x_2, vac, sol)

  Stop MPI.
- integer function, public mpi_ops::abort_mpi ()

  Abort MPI suddenly.
- integer function, public mpi_ops::broadcast_input_opts ()

  Broadcasts options (e.g. user-prescribed) that are not passed through the HDF5 output file (i.e. `ltest`, `no_plots`, ...).
- subroutine, public mpi_ops::sudden_stop (ierr)

  Suddenly stops the computations, aborting MPI, etc.

D.51 MPI_ops.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Operations related to MPI.
!-----------------------------------------------------------------------------!
module mpi_ops
#include <PB3D_macros.h>
    use str_utilities
    use messages
    use mpi
    use num_vars, only: dp, max_str_ln, pi
    use mpi_vars, only: lock_type

    implicit none
    private
    public start_mpi, stop_mpi, abort_mpi, broadcast_input_opts, &
        &sudden_stop

contains
    !> Start MPI and gather information.
    !!
    !! \return ierr
    integer function start_mpi() result(ierr)
        use num_vars, only: rank, n_procs, time_start
        use mpi_vars, only: hdf5_lock
        use input_utilities, only: pause_prog
#if ldebug
        use grid_vars, only: n_alloc_grids, n_alloc_discs
        use eq_vars, only: n_alloc_eq_1s, n_alloc_eq_2s
        use x_vars, only: n_alloc_x_1s, n_alloc_x_2s
        use sol_vars, only: n_alloc_sols
#endif

        character(*), parameter :: rout_name = 'start_MPI'

        ! initialize ierr
        ierr = 0

        ! start MPI
        call mpi_init(ierr)                                                 ! initialize MPI
        chckerr('MPI init failed')
        call mpi_comm_rank(mpi_comm_world,rank,ierr)                        ! rank
        chckerr('MPI rank failed')
        call mpi_comm_size(mpi_comm_world,n_procs,ierr)                     ! nr. processes
        chckerr('MPI size failed')

        ! create HDF5 lock
        ierr = hdf5_lock%init(10)
        chckerr('')

        ! initialize time
        call system_clock(time_start)

#if ldebug
        ! set up allocated variable counters and number of extra IO tries
        n_alloc_discs = 0
```

```
        n_alloc_grids = 0
        n_alloc_eq_1s = 0
        n_alloc_eq_2s = 0
        n_alloc_x_1s = 0
        n_alloc_x_2s = 0
        n_alloc_sols = 0
#endif
    end function start_mpi

    !> Stop MPI.
    !!
    !! Also deallocates:
    !!  - \c grid_eq
    !!  - \c grid_eq_B
    !!  - \c grid_X
    !!  - \c grid_X_B
    !!  - \c grid_sol
    !!  - \c eq_1
    !!  - \c eq_2
    !!  - \c X_1
    !!  - \c X_2
    !!  - \c sol
    !!
    !! \return ierr
    integer function stop_mpi(grid_eq,grid_eq_B,grid_X,grid_X_B,grid_sol,eq_1,&
        &eq_2,X_1,X_2,vac,sol) result(ierr)

        use mpi_vars, only: dealloc_lock, &
            &hdf5_lock
        use grid_vars, only: grid_type
        use eq_vars, only: eq_1_type, eq_2_type
        use x_vars, only: x_1_type, x_2_type
        use vac_vars, only: vac_type
        use sol_vars, only: sol_type
        use num_vars, only: eq_style
#if ldebug
        use mpi_utilities, only: get_ser_var, &
            &n_waits
        use num_vars, only: rank, n_procs
        use grid_vars, only: n_alloc_grids, n_alloc_discs
        use eq_vars, only: n_alloc_eq_1s, n_alloc_eq_2s
        use x_vars, only: n_alloc_x_1s, n_alloc_x_2s
        use sol_vars, only: n_alloc_sols
#endif

        character(*), parameter :: rout_name = 'stop_MPI'

        ! input / output
        type(grid_type), intent(inout), optional :: grid_eq                    !< equilibrium grid
        type(grid_type), intent(inout), pointer, optional :: grid_eq_b         !< field-aligned equilibrium grid
        type(grid_type), intent(inout), optional :: grid_x                     !< perturbation grid
        type(grid_type), intent(inout), pointer, optional :: grid_x_b          !< field-aligned perturbation grid
        type(grid_type), intent(inout), optional :: grid_sol                   !< solution grid
        type(eq_1_type), intent(inout), optional :: eq_1                       !< Flux equilibrium variables
        type(eq_2_type), intent(inout), optional :: eq_2                       !< metric equilibrium variables
        type(x_1_type), intent(inout), optional :: x_1                         !< vectorial perturbation variables
        type(x_2_type), intent(inout), optional :: x_2                         !< integrated tensorial perturbation variables
        type(vac_type), intent(inout), optional :: vac                         !< vacuum variables
        type(sol_type), intent(inout), optional :: sol                         !< solution variables

        ! local variables
#if ldebug
        integer :: id                                                          ! counter
        integer, allocatable :: ser_n_waits(:)                                 ! serial n_waits
#endif

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Stopping MPI')

        ! clean up variables
        if (present(grid_eq)) then
            if (associated(grid_eq%r_F)) call grid_eq%dealloc()
        end if
        if (present(grid_eq_b)) then
            if (associated(grid_eq_b)) then
                if (eq_style.eq.2) then
                    call grid_eq_b%dealloc()
                    deallocate(grid_eq_b)
                end if
                nullify(grid_eq_b)
            end if
        end if
        if (present(grid_x)) then
```

```
            if (associated(grid_x%r_F)) call grid_x%dealloc()
        end if
        if (present(grid_x_b)) then
            if (associated(grid_x_b)) then
                if (eq_style.eq.2) then
                    call grid_x_b%dealloc()
                    deallocate(grid_x_b)
                end if
                nullify(grid_x_b)
            end if
        end if
        if (present(grid_sol)) then
            if (associated(grid_sol%r_F)) call grid_sol%dealloc()
        end if
        if (present(eq_1)) then
            if (allocated(eq_1%pres_FD)) call eq_1%dealloc()
        end if
        if (present(eq_2))then
            if (allocated(eq_2%jac_FD)) call eq_2%dealloc()
        end if
        if (present(x_1)) then
            if (allocated(x_1%U_0)) call x_1%dealloc()
        end if
        if (present(x_2)) then
            if (allocated(x_2%PV_0)) call x_2%dealloc()
        end if
        if (present(vac)) then
            if (allocated(vac%res)) call vac%dealloc()
        end if
        if (present(sol)) then
            if (allocated(sol%val)) call sol%dealloc()
        end if

#if ldebug
        ! information about allocated variables
        if (n_alloc_discs.ne.0) call writo('For rank '//trim(i2str(rank))//&
            &', n_alloc_discs = '//trim(i2str(n_alloc_discs)),&
            &persistent=.true.)
        if (n_alloc_grids.ne.0) call writo('For rank '//trim(i2str(rank))//&
            &', n_alloc_grids = '//trim(i2str(n_alloc_grids)),&
            &persistent=.true.)
        if (n_alloc_eq_1s.ne.0) call writo('For rank '//trim(i2str(rank))//&
            &', n_alloc_eq_1s = '//trim(i2str(n_alloc_eq_1s)),&
            &persistent=.true.)
        if (n_alloc_eq_2s.ne.0) call writo('For rank '//trim(i2str(rank))//&
            &', n_alloc_eq_2s = '//trim(i2str(n_alloc_eq_2s)),&
            &persistent=.true.)
        if (n_alloc_x_1s.ne.0) call writo('For rank '//trim(i2str(rank))//&
            &', n_alloc_X_1s = '//trim(i2str(n_alloc_x_1s)),&
            &persistent=.true.)
        if (n_alloc_x_2s.ne.0) call writo('For rank '//trim(i2str(rank))//&
            &', n_alloc_X_2s = '//trim(i2str(n_alloc_x_2s)),&
            &persistent=.true.)
        if (n_alloc_sols.ne.0) call writo('For rank '//trim(i2str(rank))//&
            &', n_alloc_sols = '//trim(i2str(n_alloc_sols)),&
            &persistent=.true.)

        ! information about synchronization of wait_MPI
        ierr = get_ser_var([n_waits],ser_n_waits)
        chckerr('')
        if (rank.eq.0 .and. (minval(ser_n_waits).ne.n_waits .or. &
            &maxval(ser_n_waits).ne.n_waits)) then
            call writo('The number of wait_MPI commands was not consistent')
            call lvl_ud(1)
            do id = 1,n_procs
                call writo('For rank '//trim(i2str(id-1))//': '//&
                    &trim(i2str(ser_n_waits(id))))
            end do
            call lvl_ud(-1)
        end if
#endif

        ! deallocate HDF5 lock
        ierr = dealloc_lock(hdf5_lock)
        chckerr('')

        ! finalize
        call blacs_exit(1)
        call mpi_finalize(ierr)
        chckerr('MPI stop failed')
    end function stop_mpi

    !> Abort MPI suddenly.
    !!
    !! \return ierr
    integer function abort_mpi() result(ierr)
        character(*), parameter :: rout_name = 'abort_MPI'
```

```fortran

        ! initialize ierr
        ierr = 0

        call mpi_abort(mpi_comm_world,0,ierr)
        chckerr('MPI abort failed')
    end function abort_mpi

    !> Broadcasts options (e.g. user-prescribed) that are not passed through the
    !! HDF5 output file (i.e. \c ltest, \c no_plots, ...).
    !!
    !! \see read_input_opts()
    !!
    !! \note Some  variables (e.g. \c  eq_style, ...)  are not passed  over MPI.
    !! Every  process should  call  its own  reconstruct_pb3d_in()  in order  to
    !! obtain them.
    !!
    !! \return ierr
    integer function broadcast_input_opts() result(ierr)
        use num_vars, only: max_str_ln, ltest, max_it_zero, rank, &
            &max_it_rich, tol_zero, n_procs, n_sol_requested, tol_rich, &
            &max_nr_backtracks_hh, sol_n_procs, &
            &retain_all_sol, plot_flux_q, plot_magn_grid, plot_b, plot_j, &
            &plot_kappa, plot_sol_xi, plot_sol_q, plot_e_rec, no_plots, &
            &plot_grid_style, n_sol_plotted, n_theta_plot, n_zeta_plot, &
            &min_theta_plot, max_theta_plot, min_zeta_plot, max_zeta_plot, &
            &min_r_plot, max_r_plot, swap_angles, plot_resonance, tol_slepc, &
            &prog_style, post_style, jump_to_sol, export_hel, compare_tor_pos, &
            &plot_vmec_modes, ev_guess, ex_plot_style, solver_slepc_style, &
            &pert_mult_factor_post, post_output_full, post_output_sol, &
            &tol_norm, max_it_slepc, plot_vac_pot, plot_size, &
            &max_tot_mem, max_x_mem, invert_top_bottom_h, &
            &do_execute_command_line, print_mem_usage, &
            &rich_restart_lvl, &
            &pb3d_name, &
            &rz_0, &
            &min_rvac_plot, max_rvac_plot, min_zvac_plot, max_zvac_plot, &
            &n_vac_plot
        use grid_vars, only: min_par_x, max_par_x
        use rich_vars, only: no_guess, rich_lvl, min_n_par_x

        character(*), parameter :: rout_name = 'broadcast_input_opts'

        ! local variables
        character(len=max_str_ln) :: err_msg                                    ! error message

        ! initialize ierr
        ierr = 0

        ! set err_msg
        err_msg = 'MPI broadcast failed'

        if (n_procs.gt.1) then                                                  ! need to broadcast from rank 0 to other processes
            ! print message
            call writo('Broadcasting input option variables')
            call lvl_ud(1)

            ! variables that are sent for every program style:
            call mpi_bcast(ltest,1,mpi_logical,0,mpi_comm_world,ierr)
            chckerr(err_msg)
            call mpi_bcast(no_plots,1,mpi_logical,0,mpi_comm_world,ierr)
            chckerr(err_msg)
            call mpi_bcast(do_execute_command_line,1,mpi_logical,0,&
                &mpi_comm_world,ierr)
            call mpi_bcast(print_mem_usage,1,mpi_logical,0,mpi_comm_world,ierr)
            chckerr(err_msg)
            call mpi_bcast(plot_flux_q,1,mpi_logical,0,mpi_comm_world,ierr)
            chckerr(err_msg)
            call mpi_bcast(plot_kappa,1,mpi_logical,0,mpi_comm_world,ierr)
            chckerr(err_msg)
            call mpi_bcast(plot_magn_grid,1,mpi_logical,0,mpi_comm_world,ierr)
            chckerr(err_msg)
            call mpi_bcast(plot_b,1,mpi_logical,0,mpi_comm_world,ierr)
            chckerr(err_msg)
            call mpi_bcast(plot_j,1,mpi_logical,0,mpi_comm_world,ierr)
            chckerr(err_msg)
            call mpi_bcast(plot_resonance,1,mpi_logical,0,mpi_comm_world,ierr)
            chckerr(err_msg)
            call mpi_bcast(tol_zero,1,mpi_double_precision,0,mpi_comm_world,&
                &ierr)
            chckerr(err_msg)
            call mpi_bcast(min_theta_plot,1,mpi_double_precision,0,&
                &mpi_comm_world,ierr)
            chckerr(err_msg)
            call mpi_bcast(max_theta_plot,1,mpi_double_precision,0,&
                &mpi_comm_world,ierr)
```

```
            chckerr(err_msg)
            call mpi_bcast(min_zeta_plot,1,mpi_double_precision,0,&
                &mpi_comm_world,ierr)
            chckerr(err_msg)
            call mpi_bcast(max_zeta_plot,1,mpi_double_precision,0,&
                &mpi_comm_world,ierr)
            chckerr(err_msg)
            call mpi_bcast(min_r_plot,1,mpi_double_precision,0,mpi_comm_world,&
                &ierr)
            chckerr(err_msg)
            call mpi_bcast(max_r_plot,1,mpi_double_precision,0,mpi_comm_world,&
                &ierr)
            chckerr(err_msg)
            call mpi_bcast(max_tot_mem,1,mpi_double_precision,0,&
                &mpi_comm_world,ierr)
            chckerr(err_msg)
            call mpi_bcast(sol_n_procs,1,mpi_integer,0,mpi_comm_world,ierr)
            chckerr(err_msg)
            call mpi_bcast(n_theta_plot,1,mpi_integer,0,mpi_comm_world,ierr)
            chckerr(err_msg)
            call mpi_bcast(n_zeta_plot,1,mpi_integer,0,mpi_comm_world,ierr)
            chckerr(err_msg)
            call mpi_bcast(max_it_rich,1,mpi_integer,0,mpi_comm_world,ierr)
            chckerr(err_msg)
            call mpi_bcast(max_it_zero,1,mpi_integer,0,mpi_comm_world,ierr)
            chckerr(err_msg)
            call mpi_bcast(max_nr_backtracks_hh,1,mpi_integer,0,mpi_comm_world,&
                &ierr)
            chckerr(err_msg)
            call mpi_bcast(ex_plot_style,1,mpi_integer,0,mpi_comm_world,ierr)
            chckerr(err_msg)
            call mpi_bcast(plot_size,2,mpi_integer,0,mpi_comm_world,ierr)
            chckerr(err_msg)
            call mpi_bcast(pb3d_name,len(pb3d_name),mpi_character,0,&
                &mpi_comm_world,ierr)
            chckerr(err_msg)

            ! select according to program style
            select case (prog_style)
                case(1)                                                     ! PB3D
                    call mpi_bcast(no_guess,1,mpi_logical,0,mpi_comm_world,ierr)
                    chckerr(err_msg)
                    call mpi_bcast(jump_to_sol,1,mpi_logical,0,mpi_comm_world,&
                        &ierr)
                    chckerr(err_msg)
                    call mpi_bcast(export_hel,1,mpi_logical,0,mpi_comm_world,&
                        &ierr)
                    chckerr(err_msg)
                    call mpi_bcast(plot_vmec_modes,1,mpi_logical,0,&
                        &mpi_comm_world,ierr)
                    chckerr(err_msg)
                    call mpi_bcast(invert_top_bottom_h,1,mpi_logical,0,&
                        &mpi_comm_world,ierr)
                    chckerr(err_msg)
                    call mpi_bcast(min_n_par_x,1,mpi_integer,0,mpi_comm_world,&
                        &ierr)
                    chckerr(err_msg)
                    call mpi_bcast(solver_slepc_style,1,mpi_integer,0,&
                        &mpi_comm_world,ierr)
                    chckerr(err_msg)
                    call mpi_bcast(n_sol_requested,1,mpi_integer,0,&
                        &mpi_comm_world,ierr)
                    chckerr(err_msg)
                    call mpi_bcast(retain_all_sol,1,mpi_integer,0,&
                        &mpi_comm_world,ierr)
                    chckerr(err_msg)
                    call mpi_bcast(max_it_slepc,1,mpi_integer,0,mpi_comm_world,&
                        &ierr)
                    chckerr(err_msg)
                    call mpi_bcast(rich_restart_lvl,1,mpi_integer,0,&
                        &mpi_comm_world,ierr)
                    chckerr(err_msg)
                    call mpi_bcast(ev_guess,1,mpi_double_precision,0,&
                        &mpi_comm_world,ierr)
                    chckerr(err_msg)
                    call mpi_bcast(tol_norm,1,mpi_double_precision,0,&
                        &mpi_comm_world,ierr)
                    chckerr(err_msg)
                    call mpi_bcast(min_par_x,1,mpi_double_precision,0,&
                        &mpi_comm_world,ierr)
                    chckerr(err_msg)
                    call mpi_bcast(max_par_x,1,mpi_double_precision,0,&
                        &mpi_comm_world,ierr)
                    chckerr(err_msg)
                    call mpi_bcast(max_x_mem,1,mpi_double_precision,&
                        &0,mpi_comm_world,ierr)
                    chckerr(err_msg)
```

```
                    call mpi_bcast(tol_rich,1,mpi_double_precision,0,&
                        &mpi_comm_world,ierr)
                    chckerr(err_msg)
                    if (rank.ne.0) allocate(tol_slepc(max_it_rich))
                    call mpi_bcast(tol_slepc,max_it_rich,mpi_double_precision,&
                        &0,mpi_comm_world,ierr)
                    chckerr(err_msg)
                case(2)                                                 ! POST
                    call mpi_bcast(swap_angles,1,mpi_logical,0,mpi_comm_world,&
                        &ierr)
                    chckerr(err_msg)
                    call mpi_bcast(compare_tor_pos,1,mpi_logical,0,&
                        &mpi_comm_world,ierr)
                    chckerr(err_msg)
                    call mpi_bcast(plot_sol_xi,1,mpi_logical,0,mpi_comm_world,&
                        &ierr)
                    chckerr(err_msg)
                    call mpi_bcast(plot_sol_q,1,mpi_logical,0,mpi_comm_world,&
                        &ierr)
                    chckerr(err_msg)
                    call mpi_bcast(plot_vac_pot,1,mpi_logical,0,mpi_comm_world,&
                        &ierr)
                    chckerr(err_msg)
                    call mpi_bcast(plot_e_rec,1,mpi_logical,0,mpi_comm_world,&
                        &ierr)
                    chckerr(err_msg)
                    call mpi_bcast(post_output_full,1,mpi_logical,0,&
                        &mpi_comm_world,ierr)
                    chckerr(err_msg)
                    call mpi_bcast(post_output_sol,1,mpi_logical,0,&
                        &mpi_comm_world,ierr)
                    chckerr(err_msg)
                    call mpi_bcast(n_vac_plot,2,mpi_integer,0,mpi_comm_world,&
                        &ierr)
                    chckerr(err_msg)
                    call mpi_bcast(plot_grid_style,1,mpi_integer,0,&
                        &mpi_comm_world,ierr)
                    chckerr(err_msg)
                    call mpi_bcast(n_sol_plotted,4,mpi_integer,0,&
                        &mpi_comm_world,ierr)
                    chckerr(err_msg)
                    call mpi_bcast(rich_lvl,1,mpi_integer,0,mpi_comm_world,ierr)
                    chckerr(err_msg)
                    call mpi_bcast(post_style,1,mpi_integer,0,mpi_comm_world,&
                        &ierr)
                    chckerr(err_msg)
                    call mpi_bcast(pert_mult_factor_post,1,&
                        &mpi_double_precision,0,mpi_comm_world,ierr)
                    chckerr(err_msg)
                    call mpi_bcast(rz_0,2,mpi_double_precision,0,&
                        &mpi_comm_world,ierr)
                    chckerr(err_msg)
                    call mpi_bcast(min_rvac_plot,1,mpi_double_precision,0,&
                        &mpi_comm_world,ierr)
                    chckerr(err_msg)
                    call mpi_bcast(max_rvac_plot,1,mpi_double_precision,0,&
                        &mpi_comm_world,ierr)
                    chckerr(err_msg)
                    call mpi_bcast(min_zvac_plot,1,mpi_double_precision,0,&
                        &mpi_comm_world,ierr)
                    chckerr(err_msg)
                    call mpi_bcast(max_zvac_plot,1,mpi_double_precision,0,&
                        &mpi_comm_world,ierr)
                    chckerr(err_msg)
                case default
                    err_msg = 'No program style associated with '//&
                        &trim(i2str(prog_style))
                    ierr = 1
                    chckerr(err_msg)
            end select

            call lvl_ud(-1)
            call writo('Input option variables broadcasted')
        end if
    end function broadcast_input_opts

    !> Suddenly stops the computations, aborting MPI, etc.
    !!
    !! As a special case, if \c ierr = 66, no error message is printed.
    !!
    !! \return ierr
    subroutine sudden_stop(ierr)
        use num_vars, only: rank

        ! input / output
        integer, intent(in) :: ierr                                     ! error to output

```

```
        ! local variables
        integer :: ierr_abort                                                   ! error to output

        if (ierr.ne.66) then
            call writo('>> calling routine: PB3D (main) of rank '//&
                &trim(i2str(rank)),persistent=.true.)
            call writo('ERROR CODE '//trim(i2str(ierr))//&
                &'. Aborting MPI rank '//trim(i2str(rank)),&
                &persistent=.true.)
            call lvl_ud(1)
            ierr_abort = abort_mpi()
        else
            ierr_abort = stop_mpi()
        end if
        if (ierr_abort.ne.0) then
            call writo('MPI cannot abort...',persistent=.true.)
            call writo('Shutting down',persistent=.true.)
        end if
        call lvl_ud(-1)
        stop
    end subroutine
end module mpi_ops
```

D.52 /Users/toonweyens/Code/PB3D/Modules/MPI_utilities.f90 File Reference

Interfaces and Types

- interface mpi_utilities::get_ser_var

  Gather parallel variable in serial version on group master.
- interface mpi_utilities::get_ghost_arr

  Fill the ghost regions in an array.
- interface mpi_utilities::broadcast_var

  Wrapper function to broadcast a single variable using MPI.

Modules

- module mpi_utilities

  Numerical utilities related to MPI.

Functions/Subroutines

- integer function, public mpi_utilities::redistribute_var (var, dis_var, lims, lims_dis)

  Redistribute variables according to new limits.
- integer function, public mpi_utilities::wait_mpi ()

  Wait for all processes, wrapper to MPI barrier.
- integer function, public mpi_utilities::lock_req_acc (lock, blocking)

  Request access to lock of a BL (blocking) or optionally a NB (non-blocking) type.
- integer function, public mpi_utilities::lock_return_acc (lock)

  Returns access to a lock.
- integer function, public mpi_utilities::lock_wl_change (wl_action, blocking, lock_loc, wl, ranks)

  Adds, removes or sets to active a rank from the waiting list for a lock and returns the lock waiting list:
- character(len=max_str_ln) function, public mpi_utilities::lock_header (lock_loc)

  Returns the header for lock debug messages.

Variables

- logical, public mpi_utilities::debug_lock = .false.

  print debug information about lock operations
- integer, public mpi_utilities::n_waits = 0

  number of waits

D.53 MPI_utilities.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------!
!> Numerical utilities related to MPI.
!!
!! This  includes  a  lock  system,  which  can be  both  BL  (blocking)  or  NB
!! (non-blocking). It  is based on the  implementation of an MPI-IO  atomic mode
!! without file support, described in \cite RossAtomicIO.
!!
!! \see See mpi_vars.
!!
!! The reason  for this was the fact  that using a simple lock file  can lead to
!! crashes.
!!
!! \note A  downside of this method  is that in  some rare cases a  deadlock may
!! occur as the master process, which contains the shared variable with a window
!! that other  processes may use,  is idle and  waiting, whereas the  others are
!! still performing  lock options. As  the master is  idle and waiting,  its MPI
!! asynchronous  communication  is not  performed.  To  remedy this,  just  call
!! wait_MPI() after procedures where lock operations are performed.
!-----------------------------------------------------------------------!
module mpi_utilities
#include <PB3D_macros.h>
    use str_utilities
    use messages
    use mpi
    use num_vars, only: dp, max_str_ln, pi
    use mpi_vars, only: lock_type

    implicit none
    private
    public get_ser_var, redistribute_var, get_ghost_arr, broadcast_var, &
        &wait_mpi, lock_req_acc, lock_return_acc
#if ldebug
    public lock_header, lock_wl_change, &
        &debug_lock, n_waits
#endif

#if ldebug
    ! global variables
    ! Note: use "sort -nk1 [output file]" to get output sorted by time.
    logical :: debug_lock = .false.                                         !< print debug information about lock operations \ldebug
    integer :: n_waits = 0                                                  !< number of waits \ldebug
#endif

    ! interfaces

    !> \public Gather parallel variable in serial version on group master.
    !!
    !! Optionally,  all the  processes receive  the parallel  variable using  \c
    !! scatter.
    !!
    !! \note The  serial variable has to  be allocatable and if  unallocated, it
    !! will be allocated.
    !!
    !! \return ierr
    interface get_ser_var
        !> \public
        module procedure get_ser_var_complex
        !> \public
        module procedure get_ser_var_real
        !> \public
        module procedure get_ser_var_int
    end interface

    !> \public Fill the ghost regions in an array.
    !!
```

```
    !! This is done by  sending the first normal point of a  process to the left
    !! process.
    !!
    !! Every MPI message is identified by  its sending process. The array should
    !! have the extended size, including ghost regions.
    !!
    !! \return ierr
    interface get_ghost_arr
        !> \public
        module procedure get_ghost_arr_3d_complex
        !> \public
        module procedure get_ghost_arr_3d_real
        !> \public
        module procedure get_ghost_arr_2d_complex
        !> \public
        module procedure get_ghost_arr_1d_real
    end interface

    !> \public Wrapper function to broadcast a single variable using MPI.
    !!
    !! \return ierr
    interface broadcast_var
        !> \public
        module procedure broadcast_var_real
        !> \public
        module procedure broadcast_var_int
        !> \public
        module procedure broadcast_var_log
        !> \public
        module procedure broadcast_var_complex_arr
        !> \public
        module procedure broadcast_var_real_arr
        !> \public
        module procedure broadcast_var_int_arr
        !> \public
        module procedure broadcast_var_log_arr
    end interface

contains
    !> \private complex version
    integer function get_ser_var_complex(var,ser_var,scatter) result(ierr)
        use num_vars, only: rank, n_procs

        character(*), parameter :: rout_name = 'get_ser_var'

        ! input / output
        complex(dp), intent(in) :: var(:)                                   !< parallel vector
        complex(dp), allocatable, intent(inout) :: ser_var(:)               !< serial vector
        logical, intent(in), optional :: scatter                            !< optionally scatter the result to all the processes

        ! local variables
        character(len=max_str_ln) :: err_msg                                ! error message
        integer, allocatable :: recvcounts(:)                               ! counts of nr. of elements received from each processor
        integer, allocatable :: displs(:)                                   ! displacements elements received from each processor
        integer :: id                                                       ! counter
        logical :: scatter_loc                                              ! local copy of scatter

        ! initialize ierr
        ierr = 0

        ! set local copy of scatter
        scatter_loc = .false.
        if (present(scatter)) scatter_loc = scatter

        ! gather local size  of var of all groups onto  main processor, to serve
        ! as receive counts on group master
        if (rank.eq.0 .or. scatter_loc) then
            allocate(recvcounts(n_procs))
            allocate(displs(n_procs))
        else
            allocate(recvcounts(0))
            allocate(displs(0))
        end if
        if (scatter_loc) then
            call mpi_allgather(size(var),1,mpi_integer,recvcounts,1,&
                &mpi_integer,mpi_comm_world,ierr)
        else
            call mpi_gather(size(var),1,mpi_integer,recvcounts,1,&
                &mpi_integer,0,mpi_comm_world,ierr)
        end if
        err_msg = 'Failed to gather size of parallel variable'
        chckerr(err_msg)

        ! allocate serial variable
```

```
        if (allocated(ser_var)) then
            if (size(ser_var).ne.sum(recvcounts)) then
                ierr = 1
                write(*,*) size(ser_var), sum(recvcounts)
                err_msg = 'ser_var has wrong dimensions'
                chckerr(err_msg)
            end if
        else
            allocate(ser_var(sum(recvcounts)))
        end if

        ! deduce displacements by summing recvcounts
        if (rank.eq.0 .or. scatter_loc) then
            displs(1) = 0
            do id = 2,n_procs
                displs(id) = displs(id-1) + recvcounts(id-1)
            end do
        end if

        if (scatter_loc) then
            call mpi_allgatherv(var,size(var),mpi_double_complex,ser_var,&
                &recvcounts,displs,mpi_double_complex,mpi_comm_world,ierr)
        else
            call mpi_gatherv(var,size(var),mpi_double_complex,ser_var,&
                &recvcounts,displs,mpi_double_complex,0,mpi_comm_world,ierr)
        end if
        err_msg = 'Failed to gather parallel variable'
        chckerr(err_msg)
    end function get_ser_var_complex
    !> \private real version
    integer function get_ser_var_real(var,ser_var,scatter) result(ierr)
        use num_vars, only: rank, n_procs

        character(*), parameter :: rout_name = 'get_ser_var_real'

        ! input / output
        real(dp), intent(in) :: var(:)                                  !< parallel vector
        real(dp), allocatable, intent(inout) :: ser_var(:)              !< serial vector
        logical, intent(in), optional :: scatter                        !< optionally scatter the result to all the processes

        ! local variables
        character(len=max_str_ln) :: err_msg                            ! error message
        integer, allocatable :: recvcounts(:)                           ! counts of nr. of elements received from each processor
        integer, allocatable :: displs(:)                               ! displacements elements received from each processor
        integer :: id                                                   ! counter
        logical :: scatter_loc                                          ! local copy of scatter

        ! initialize ierr
        ierr = 0

        ! set local copy of scatter
        scatter_loc = .false.
        if (present(scatter)) scatter_loc = scatter

        ! gather local size  of var of all groups onto  main processor, to serve
        ! as receive counts on group master
        if (rank.eq.0 .or. scatter_loc) then
            allocate(recvcounts(n_procs))
            allocate(displs(n_procs))
        else
            allocate(recvcounts(0))
            allocate(displs(0))
        end if
        if (scatter_loc) then
            call mpi_allgather(size(var),1,mpi_integer,recvcounts,1,&
                &mpi_integer,mpi_comm_world,ierr)
        else
            call mpi_gather(size(var),1,mpi_integer,recvcounts,1,&
                &mpi_integer,0,mpi_comm_world,ierr)
        end if
        err_msg = 'Failed to gather size of parallel variable'
        chckerr(err_msg)

        ! allocate serial variable
        if (allocated(ser_var)) then
            if (size(ser_var).ne.sum(recvcounts)) then
                ierr = 1
                write(*,*) size(ser_var), sum(recvcounts)
                err_msg = 'ser_var has wrong dimensions'
                chckerr(err_msg)
            end if
        else
            allocate(ser_var(sum(recvcounts)))
        end if
```

```

        ! deduce displacements by summing recvcounts
        if (rank.eq.0 .or. scatter_loc) then
            displs(1) = 0
            do id = 2,n_procs
                displs(id) = displs(id-1) + recvcounts(id-1)
            end do
        end if

        if (scatter_loc) then
            call mpi_allgatherv(var,size(var),mpi_double_precision,ser_var,&
                &recvcounts,displs,mpi_double_precision,mpi_comm_world,ierr)
        else
            call mpi_gatherv(var,size(var),mpi_double_precision,ser_var,&
                &recvcounts,displs,mpi_double_precision,0,mpi_comm_world,ierr)
        end if
        err_msg = 'Failed to gather parallel variable'
        chckerr(err_msg)
    end function get_ser_var_real
    !> \private integer version
    integer function get_ser_var_int(var,ser_var,scatter) result(ierr)
        use num_vars, only: rank, n_procs

        character(*), parameter :: rout_name = 'get_ser_var_int'

        ! input / output
        integer, intent(in) :: var(:)                                   !< parallel vector
        integer, allocatable, intent(inout) :: ser_var(:)               !< serial vector
        logical, intent(in), optional :: scatter                        !< optionally scatter the result to all the processes

        ! local variables
        character(len=max_str_ln) :: err_msg                            ! error message
        integer, allocatable :: recvcounts(:)                           ! counts of nr. of elements received from each processor
        integer, allocatable :: displs(:)                               ! displacements elements received from each processor
        integer :: id                                                   ! counter
        logical :: scatter_loc                                          ! local copy of scatter

        ! initialize ierr
        ierr = 0

        ! set local copy of scatter
        scatter_loc = .false.
        if (present(scatter)) scatter_loc = scatter

        ! gather local size  of var of all groups onto  main processor, to serve
        ! as receive counts on group master
        if (rank.eq.0 .or. scatter_loc) then
            allocate(recvcounts(n_procs))
            allocate(displs(n_procs))
        else
            allocate(recvcounts(0))
            allocate(displs(0))
        end if
        if (scatter_loc) then
            call mpi_allgather(size(var),1,mpi_integer,recvcounts,1,&
                &mpi_integer,mpi_comm_world,ierr)
        else
            call mpi_gather(size(var),1,mpi_integer,recvcounts,1,&
                &mpi_integer,0,mpi_comm_world,ierr)
        end if
        err_msg = 'Failed to gather size of parallel variable'
        chckerr(err_msg)

        ! allocate serial variable
        if (allocated(ser_var)) then
            if (size(ser_var).ne.sum(recvcounts)) then
                ierr = 1
                write(*,*) size(ser_var), sum(recvcounts)
                err_msg = 'ser_var has wrong dimensions'
                chckerr(err_msg)
            end if
        else
            allocate(ser_var(sum(recvcounts)))
        end if

        ! deduce displacements by summing recvcounts
        if (rank.eq.0 .or. scatter_loc) then
            displs(1) = 0
            do id = 2,n_procs
                displs(id) = displs(id-1) + recvcounts(id-1)
            end do
        end if

        if (scatter_loc) then
```

```
            call mpi_allgatherv(var,size(var),mpi_integer,ser_var,&
                &recvcounts,displs,mpi_integer,mpi_comm_world,ierr)
        else
            call mpi_gatherv(var,size(var),mpi_integer,ser_var,&
                &recvcounts,displs,mpi_integer,0,mpi_comm_world,ierr)
        end if
        err_msg = 'Failed to gather parallel variable'
        chckerr(err_msg)
    end function get_ser_var_int

    !> Redistribute variables according to new limits.
    integer function redistribute_var(var,dis_var,lims,lims_dis) result(ierr)
        use num_vars, only: n_procs, rank

        character(*), parameter :: rout_name = 'redistribute_var'

        ! input / output
        real(dp), intent(in) :: var(:)                                      !< parallel vector
        real(dp), intent(inout) :: dis_var(:)                               !< redistributed vector
        integer, intent(in) :: lims(2)                                      !< indices of parallel vector
        integer, intent(in) :: lims_dis(2)                                  !< indices of redistributed parallel vector

        ! local variables
        integer :: id, jd                                                   ! counters
        integer, allocatable :: lims_tot(:,:)                               ! limits of all processes
        integer, allocatable :: lims_dis_tot(:,:)                           ! redistributed limits of all processes
        integer, allocatable :: temp_lim(:)                                 ! temporary variable
        integer, allocatable :: n_vars(:,:)                                 ! number of values to be received and sent from each process (row: to, column: from)
        integer, allocatable :: nr_sen(:)                                   ! number of values to be sent to each process from this process
        integer, allocatable :: nr_rec(:)                                   ! number of values to be received from each process to this process
        integer, allocatable :: id_rec(:)                                   ! starting position of variable to be received in local memory (starting at 0)
        integer, allocatable :: id_sen(:)                                   ! starting position of variable to be sent in local memory (starting at 0)
        integer :: max_loc                                                  ! local maximum

        ! initialize ierr
        ierr = 0

        ! get limits and redistributed limits of all procs
        allocate(lims_tot(2,n_procs))
        allocate(lims_dis_tot(2,n_procs))
        do id = 1,2
            ierr = get_ser_var(lims(id:id),temp_lim,scatter=.true.)
            chckerr('')
            lims_tot(id,:) = temp_lim
            ierr = get_ser_var(lims_dis(id:id),temp_lim,scatter=.true.)
            chckerr('')
            lims_dis_tot(id,:) = temp_lim
        end do

        ! find  out how many  variables to send and  receive. A row  indicates a
        ! receive and a column a send. E.g.:
        !   (a b)
        !   (c d)
        ! means that  process 0  gets a  from process  0 and  b from  process 1.
        ! Process 1 gets c from process 0 and d from process 1.
        allocate(n_vars(n_procs,n_procs))
        n_vars = 0
        do jd = 1,n_procs
            max_loc = lims_dis_tot(1,jd)
            do id = 1,n_procs
                if (lims_tot(2,id).ge.max_loc) then
                    n_vars(jd,id) = min(lims_tot(2,id),lims_dis_tot(2,jd)) - &
                        &max_loc + 1
                    max_loc = max_loc + n_vars(jd,id)
                end if
            end do
        end do

        ! set up how many are to be sent by each process and which memory index
        !   - nr_rec is the row of n_vars corresponding to this process,
        !   - nr_sen is the column of n_vars corresponding to this process,
        !   -  id_rec  counts  the  number of  variables  gotten  from  previous
        !   processes,
        !   - id_sen counts the number  of variables sent to previous processes,
        !   taking into account  the difference betweeen the lower  limit of the
        !   process sending to and the current process.
        allocate(nr_rec(n_procs))
        allocate(nr_sen(n_procs))
        allocate(id_rec(n_procs))
        allocate(id_sen(n_procs))
        do id = 1,n_procs
            nr_rec(id) = n_vars(rank+1,id)
```

```
            nr_sen(id) = n_vars(id,rank+1)
            id_rec(id) = sum(n_vars(rank+1,1:id-1))
            id_sen(id) = sum(n_vars(id,1:rank)) + lims_dis_tot(1,id) - lims(1)
        end do

        call mpi_alltoallv(var,nr_sen,id_sen,mpi_double_precision,&
            &dis_var,nr_rec,id_rec,mpi_double_precision,mpi_comm_world,ierr)
        chckerr('')

        !call sleep(rank)
        !write(*,*) 'rank', rank
        !write(*,*) '    nr_rec', nr_rec
        !write(*,*) '    id_rec', id_rec
        !write(*,*) '    nr_sen', nr_sen
        !write(*,*) '    id_sen', id_sen
        !do id = 1,n_procs
            !write(*,*) '    ', n_vars(id,:)
        !end do
    end function redistribute_var

    !> \private 3-D complex version
    integer function get_ghost_arr_3d_complex(arr,size_ghost) result(ierr)
        use num_vars, only: rank, n_procs

        character(*), parameter :: rout_name = 'get_ghost_arr_3D_complex'

        ! input / output
        complex(dp), intent(inout) :: arr(:,:,:)                                !< divided array
        integer, intent(in) :: size_ghost                                       !< width of ghost region

        ! local variables
        integer :: n_modes(2)                                                   ! number of modes
        integer :: tot_size                                                     ! total size (including ghost region)
        integer :: istat(mpi_status_size)                                       ! status of send-receive

        ! initialize ierr
        ierr = 0

        ! initialize number of modes and total size
        n_modes = [size(arr,1),size(arr,2)]
        tot_size = size(arr,3)

        ! ghost regions only make sense if there is more than 1 process
        if (n_procs.gt.1) then
            if (rank.eq.0) then                                                 ! first rank only receives
                call mpi_recv(arr(:,:,tot_size-size_ghost+1:tot_size),&
                    &size_ghost*product(n_modes),mpi_double_complex,rank+1,&
                    &rank+1,mpi_comm_world,istat,ierr)
                chckerr('Failed to receive')
            else if (rank+1.eq.n_procs) then                                    ! last rank only sends
                call mpi_send(arr(:,:,1:size_ghost),&
                    &size_ghost*product(n_modes),mpi_double_complex,rank-1,&
                    &rank,mpi_comm_world,ierr)
                chckerr('Failed to send')
            else                                                                ! middle ranks send and receive
                call mpi_sendrecv(arr(:,:,1:size_ghost),&
                    &size_ghost*product(n_modes),mpi_double_complex,rank-1,&
                    &rank,arr(:,:,tot_size-size_ghost+1:tot_size),&
                    &size_ghost*product(n_modes),mpi_double_complex,&
                    &rank+1,rank+1,mpi_comm_world,istat,ierr)
                chckerr('Failed to send and receive')
            end if
        end if
    end function get_ghost_arr_3d_complex
    !> \private 3-D real version
    integer function get_ghost_arr_3d_real(arr,size_ghost) result(ierr)
        use num_vars, only: rank, n_procs

        character(*), parameter :: rout_name = 'get_ghost_arr_3D_real'

        ! input / output
        real(dp), intent(inout) :: arr(:,:,:)                                   !< divided array
        integer, intent(in) :: size_ghost                                       !< width of ghost region

        ! local variables
        integer :: n_modes(2)                                                   ! number of modes
        integer :: tot_size                                                     ! total size (including ghost region)
        integer :: istat(mpi_status_size)                                       ! status of send-receive

        ! initialize ierr
        ierr = 0

        ! initialize number of modes and total size
        n_modes = [size(arr,1),size(arr,2)]
        tot_size = size(arr,3)

        ! ghost regions only make sense if there is more than 1 process
```

```
        if (n_procs.gt.1) then
            if (rank.eq.0) then                                         ! first rank only receives
                call mpi_recv(arr(:,:,tot_size-size_ghost+1:tot_size),&
                    &size_ghost*product(n_modes),mpi_double_precision,&
                    &rank+1,rank+1,mpi_comm_world,istat,ierr)
                chckerr('Failed to receive')
            else if (rank+1.eq.n_procs) then                            ! last rank only sends
                call mpi_send(arr(:,:,1:size_ghost),&
                    &size_ghost*product(n_modes),mpi_double_precision,&
                    &rank-1,rank,mpi_comm_world,ierr)
                chckerr('Failed to send')
            else                                                        ! middle ranks send and receive
                call mpi_sendrecv(arr(:,:,1:size_ghost),&
                    &size_ghost*product(n_modes),mpi_double_precision,&
                    &rank-1,&
                    &rank,arr(:,:,tot_size-size_ghost+1:tot_size),&
                    &size_ghost*product(n_modes),mpi_double_precision,&
                    &rank+1,rank+1,mpi_comm_world,istat,ierr)
                chckerr('Failed to send and receive')
            end if
        end if
    end function get_ghost_arr_3d_real
    !> \private 2-D complex version
    integer function get_ghost_arr_2d_complex(arr,size_ghost) result(ierr)
        use num_vars, only: rank, n_procs

        character(*), parameter :: rout_name = 'get_ghost_arr_2D_complex'

        ! input / output
        complex(dp), intent(inout) :: arr(:,:)                          !< divided array
        integer, intent(in) :: size_ghost                               !< width of ghost region

        ! local variables
        integer :: n_modes                                              ! number of modes
        integer :: tot_size                                             ! total size (including ghost region)
        integer :: istat(mpi_status_size)                               ! status of send-receive

        ! initialize ierr
        ierr = 0

        ! initialize number of modes and total size
        n_modes = size(arr,1)
        tot_size = size(arr,2)

        ! ghost regions only make sense if there is more than 1 process
        if (n_procs.gt.1) then
            if (rank.eq.0) then                                         ! first rank only receives
                call mpi_recv(arr(:,tot_size-size_ghost+1:tot_size),&
                    &size_ghost*n_modes,mpi_double_complex,rank+1,&
                    &rank+1,mpi_comm_world,istat,ierr)
                chckerr('Failed to receive')
            else if (rank+1.eq.n_procs) then                            ! last rank only sends
                call mpi_send(arr(:,1:size_ghost),size_ghost*n_modes,&
                    &mpi_double_complex,rank-1,rank,mpi_comm_world,&
                    &ierr)
                chckerr('Failed to send')
            else                                                        ! middle ranks send and receive
                call mpi_sendrecv(arr(:,1:size_ghost),size_ghost*n_modes,&
                    &mpi_double_complex,rank-1,rank,&
                    &arr(:,tot_size-size_ghost+1:tot_size),size_ghost*n_modes,&
                    &mpi_double_complex,rank+1,rank+1,mpi_comm_world,&
                    &istat,ierr)
                chckerr('Failed to send and receive')
            end if
        end if
    end function get_ghost_arr_2d_complex
    !> \private 1-D real version
    integer function get_ghost_arr_1d_real(arr,size_ghost) result(ierr)
        use num_vars, only: rank, n_procs

        character(*), parameter :: rout_name = 'get_ghost_arr_1D_real'

        ! input / output
        real(dp), intent(in) :: arr(:)                                  !< divided array
        integer, intent(in) :: size_ghost                               !< width of ghost region

        ! local variables
        integer :: tot_size                                             ! total size (including ghost region)
        integer :: istat(mpi_status_size)                               ! status of send-receive

        ! initialize ierr
        ierr = 0

        ! initialize number of modes and total size
        tot_size = size(arr)

        ! ghost regions only make sense if there is more than 1 process
```

```
        if (n_procs.gt.1) then
            if (rank.eq.0) then                                         ! first rank only receives
                call mpi_recv(arr(tot_size-size_ghost+1:tot_size),&
                    &size_ghost,mpi_double_precision,rank+1,&
                    &rank+1,mpi_comm_world,istat,ierr)
                chckerr('Failed to receive')
            else if (rank+1.eq.n_procs) then                            ! last rank only sends
                call mpi_send(arr(1:size_ghost),size_ghost,&
                    &mpi_double_precision,rank-1,rank,mpi_comm_world,&
                    &ierr)
                chckerr('Failed to send')
            else                                                        ! middle ranks send and receive
                call mpi_sendrecv(arr(1:size_ghost),size_ghost,&
                    &mpi_double_precision,rank-1,rank,&
                    &arr(tot_size-size_ghost+1:tot_size),size_ghost,&
                    &mpi_double_precision,rank+1,rank+1,&
                    &mpi_comm_world,istat,ierr)
                chckerr('Failed to send and receive')
            end if
        end if
    end function get_ghost_arr_1d_real

    !> \private real version
    integer function broadcast_var_real(var,source) result(ierr)
        character(*), parameter :: rout_name = 'broadcast_var_real'

        ! input / output
        real(dp), intent(in) :: var                                     !< variable to be broadcast
        integer, intent(in), optional :: source                         !< process that sends

        ! local variables
        integer :: source_loc = 0                                       ! local value for source

        ! initialize ierr
        ierr = 0

        ! set local source if given
        if (present(source)) source_loc = source

        call mpi_bcast(var,1,mpi_double_precision,source_loc,mpi_comm_world,&
            &ierr)
        chckerr('MPI broadcast failed')
    end function broadcast_var_real
    !> \private integer version
    integer function broadcast_var_int(var,source) result(ierr)
        character(*), parameter :: rout_name = 'broadcast_var_int'

        ! input / output
        integer, intent(in) :: var                                      !< variable to be broadcast
        integer, intent(in), optional :: source                         !< process that sends

        ! local variables
        integer :: source_loc = 0                                       ! local value for source

        ! initialize ierr
        ierr = 0

        ! set local source if given
        if (present(source)) source_loc = source

        call mpi_bcast(var,1,mpi_integer,source_loc,mpi_comm_world,ierr)
        chckerr('MPI broadcast failed')
    end function broadcast_var_int
    !> \private logical version
    integer function broadcast_var_log(var,source) result(ierr)
        character(*), parameter :: rout_name = 'broadcast_var_log'

        ! input / output
        logical, intent(in) :: var                                      !< variable to be broadcast
        integer, intent(in), optional :: source                         !< process that sends

        ! local variables
        integer :: source_loc = 0                                       ! local value for source

        ! initialize ierr
        ierr = 0

        ! set local source if given
        if (present(source)) source_loc = source

        call mpi_bcast(var,1,mpi_logical,source_loc,mpi_comm_world,ierr)
        chckerr('MPI broadcast failed')
    end function broadcast_var_log
    !> \private complex array version
    integer function broadcast_var_complex_arr(var,source) result(ierr)
        character(*), parameter :: rout_name = 'broadcast_var_complex_arr'

```

```
        ! input / output
        complex(dp), intent(in) :: var(:)                                   !< variable to be broadcast
        integer, intent(in), optional :: source                             !< process that sends

        ! local variables
        integer :: source_loc = 0                                           ! local value for source

        ! initialize ierr
        ierr = 0

        ! set local source if given
        if (present(source)) source_loc = source

        call mpi_bcast(var,size(var),mpi_double_complex,source_loc,&
            &mpi_comm_world,ierr)
        chckerr('MPI broadcast failed')
    end function broadcast_var_complex_arr
    !> \private real array version
    integer function broadcast_var_real_arr(var,source) result(ierr)
        character(*), parameter :: rout_name = 'broadcast_var_real_arr'

        ! input / output
        real(dp), intent(in) :: var(:)                                      !< variable to be broadcast
        integer, intent(in), optional :: source                             !< process that sends

        ! local variables
        integer :: source_loc = 0                                           ! local value for source

        ! initialize ierr
        ierr = 0

        ! set local source if given
        if (present(source)) source_loc = source

        call mpi_bcast(var,size(var),mpi_double_precision,source_loc,&
            &mpi_comm_world,ierr)
        chckerr('MPI broadcast failed')
    end function broadcast_var_real_arr
    !> \private integer array version
    integer function broadcast_var_int_arr(var,source) result(ierr)
        character(*), parameter :: rout_name = 'broadcast_var_int_arr'

        ! input / output
        integer, intent(in) :: var(:)                                       !< variable to be broadcast
        integer, intent(in), optional :: source                             !< process that sends

        ! local variables
        integer :: source_loc = 0                                           ! local value for source

        ! initialize ierr
        ierr = 0

        ! set local source if given
        if (present(source)) source_loc = source

        call mpi_bcast(var,size(var),mpi_integer,source_loc,mpi_comm_world,ierr)
        chckerr('MPI broadcast failed')
    end function broadcast_var_int_arr
    !> \private logical array version
    integer function broadcast_var_log_arr(var,source) result(ierr)
        character(*), parameter :: rout_name = 'broadcast_var_log_arr'

        ! input / output
        logical, intent(in) :: var(:)                                       !< variable to be broadcast
        integer, intent(in), optional :: source                             !< process that sends

        ! local variables
        integer :: source_loc = 0                                           ! local value for source

        ! initialize ierr
        ierr = 0

        ! set local source if given
        if (present(source)) source_loc = source

        call mpi_bcast(var,size(var),mpi_logical,source_loc,mpi_comm_world,ierr)
        chckerr('MPI broadcast failed')
    end function broadcast_var_log_arr

    !> Wait for all processes, wrapper to MPI barrier.
    !!
    !! \return ierr
    integer function wait_mpi() result(ierr)
        character(*), parameter :: rout_name = 'wait_MPI'

        ! initialize ierr
        ierr = 0
```

```fortran

        ! set barrier
        call mpi_barrier(mpi_comm_world,ierr)
        chckerr('MPI Barrier failed')

#if ldebug
        n_waits = n_waits + 1
#endif
    end function wait_mpi

    !> Request access to lock of a BL (blocking) or optionally a NB
    !! (non-blocking) type.
    !!
    !! \note Based on \cite RossAtomicIO.
    !!
    !! \return ierr
    integer function lock_req_acc(lock,blocking) result(ierr)
        use num_vars, only: rank
        use num_utilities, only: bubble_sort

        character(*), parameter :: rout_name = 'lock_req_acc'

        ! input / output
        type(lock_type), intent(inout) :: lock                                  !< lock

        ! local variables
        integer :: id                                                           ! counter
        integer, allocatable :: wl_loc(:)                                       ! local copy of waiting list
        integer, allocatable :: next_nb_procs(:)                                ! next NB process(es)
        integer, allocatable :: ranks_to_activate(:)                            ! ranks to be set active
        logical, intent(in), optional :: blocking                               ! is a blocking process
        logical :: next_nb_proc_exists                                          ! a next NB process exists
        logical :: direct_receipt                                               ! receipt was direct
#if ldebug
        integer :: istat                                                        ! status
#endif

        ! initialize ierr
        ierr = 0

#if ldebug
        ! check for initialized lock
        if (.not.allocated(lock%wl)) then
            ierr = 1
            chckerr('lock not intialized')
        end if
#endif

        ! set blocking property
        lock%blocking = .true.
        if (present(blocking)) lock%blocking = blocking

#if ldebug
        if(debug_lock) write(*,*,iostat=istat) trim(lock_header(lock)), &
            &'requesting access'
#endif

        ! add current process to waiting list
        ierr = lock_wl_change(1,lock%blocking,lock,wl_loc)
        chckerr('')

        ! set direct_receipt when wait list empty
        direct_receipt = wl_empty(wl_loc,[-2,-1,1,2])

        ! wait for notification if other processes
        if (direct_receipt) then
#if ldebug
            if(debug_lock) write(*,*,iostat=istat) trim(lock_header(lock)), &
                &'and got it right away'
#endif

            ! find next NB if this one is NB also
            next_nb_proc_exists = .false.
            if (.not.lock%blocking) then
                ! find all NB procs
                next_nb_proc_exists =  &
                    &.not.wl_empty(wl_loc,[-1],next_procs=next_nb_procs)
#if ldebug
                if (debug_lock) write(*,*,iostat=istat) &
                    &trim(lock_header(lock)),&
                    &'    NB proc, next NB procs found:', next_nb_procs
#endif
            end if

            ! set ranks to activate
            if (.not.lock%blocking .and. next_nb_proc_exists) then              ! more processes
                allocate(ranks_to_activate(size(next_nb_procs)+1))
```

```
                ranks_to_activate = [rank,next_nb_procs]
            else                                                        ! only this process
                allocate(ranks_to_activate(1))
                ranks_to_activate = rank
            end if
            call bubble_sort(ranks_to_activate)                         ! sort

            ! activate
#if ldebug
            if(debug_lock) write(*,*,iostat=istat) trim(lock_header(lock)), &
                &'    setting status to activate:', ranks_to_activate
#endif
            ierr = lock_wl_change(2,.false.,lock,wl_loc,ranks=ranks_to_activate)
            chckerr('')

            ! notify other ranks to activate (NB)
            if (next_nb_proc_exists) then
                do id = 1,size(next_nb_procs)
                    ierr = lock_notify(lock,next_nb_procs(id))
                    chckerr('')
                end do
            end if
        else
            ! wait to get notified
            ierr = lock_get_notified(lock)
            chckerr('')
        end if
    end function lock_req_acc

    !> Returns  access  to a  lock.
    !!
    !! The blocking property has been set when requesting the lock.
    !!
    !! \note Based on \cite RossAtomicIO.
    !!
    !! \return ierr
    integer function lock_return_acc(lock) result(ierr)
        use num_utilities, only: bubble_sort

        character(*), parameter :: rout_name = 'lock_return_acc'

        ! input / output
        type(lock_type), intent(inout) :: lock                          !< lock

        ! local variables
        integer :: id                                                   ! counter
        integer, allocatable :: wl_loc(:)                               ! local copy of waiting list
        integer, allocatable :: next_procs(:)                           ! next process(es)
        integer, allocatable :: ranks_to_activate(:)                    ! ranks to be set active
        logical :: next_proc_exists                                     ! a next process exists
        logical :: next_proc_bl                                         ! next process is BL
#if ldebug
        integer :: istat                                                ! status
#endif

        ! initialize ierr
        ierr = 0

#if ldebug
        if (.not.allocated(lock%wl)) then
            ierr = 1
            chckerr('lock not intialized')
        end if

        if(debug_lock) write(*,*,iostat=istat) trim(lock_header(lock)), &
            &'returning access'
#endif

        ! remove current process to lock waiting list
        ierr = lock_wl_change(0,lock%blocking,lock,wl_loc)
        chckerr('')

        ! find next processes if:
        !   - BL: always
        !   - NB: was last NB running
        next_proc_bl = .false.
        next_proc_exists = .false.
        if (lock%blocking .or. wl_empty(wl_loc,[-2])) then
            ! find all BL procs
            next_proc_exists = .not.wl_empty(wl_loc,[1],next_procs=next_procs)
            if (next_proc_exists) next_proc_bl = .true.
#if ldebug
            if (debug_lock) write(*,*,iostat=istat) trim(lock_header(lock)),&
                &'    next BL procs found:', next_procs
#endif

            ! if none found, try NB procs
```

```
            if (.not. next_proc_exists) then
                next_proc_exists = .not.wl_empty(wl_loc,[-1],&
                    &next_procs=next_procs)
                if (next_proc_exists) next_proc_bl = .false.
#if ldebug
                if (debug_lock) write(*,*,iostat=istat) &
                    &trim(lock_header(lock)),&
                    &'    next NB procs found:', next_procs
#endif
            end if
#if ldebug
        else
            if(debug_lock) write(*,*,iostat=istat) trim(lock_header(lock)), &
                &'but has no notification rights'
#endif
        end if

        if (next_proc_exists) then
            ! set ranks to activate
            if (next_proc_bl) then                                          ! only the BL process
                allocate(ranks_to_activate(1))
                ranks_to_activate = next_procs(1)
            else                                                            ! multiple NB processes
                allocate(ranks_to_activate(size(next_procs)))
                ranks_to_activate = next_procs
            end if
            call bubble_sort(ranks_to_activate)                             ! sort

            ! activate
#if ldebug
            if(debug_lock) write(*,*,iostat=istat) trim(lock_header(lock)), &
                &'    setting status to activate:', ranks_to_activate
#endif
            ierr = lock_wl_change(2,next_proc_bl,lock,wl_loc,&
                &ranks=ranks_to_activate)
            chckerr('')

            ! notify ranks to activate (NB or BL)
            if (next_proc_exists) then
                do id = 1,size(ranks_to_activate)
                    ierr = lock_notify(lock,ranks_to_activate(id))
                    chckerr('')
                end do
            end if
        end if
    end function lock_return_acc

    !> Decides whether a waiting list is empty.
    !!
    !! The type of process to find is  indicated by an array of possible values.
    !!
    !! \see See lock_wl_change() for an explanation of the process type.
    !!
    !! Additionally, for NB processes, the  negative inverse of these values are
    !! used.
    !!
    !! If the waiting list is not  empty, the next process(es) can optionally be
    !! returned.
    !!
    !! \note Based on \cite RossAtomicIO.
    !!
    !! \return ierr
    logical function wl_empty(wl,proc_type,next_procs)
        use num_vars, only: n_procs, rank

        ! input / output
        integer, intent(in) :: wl(:)                                        !< waiting list
        integer, intent(in) :: proc_type(:)                                 !< types of processes accepted
        integer, intent(inout), optional, allocatable :: next_procs(:)      !< next process(es) if not empty

        ! local variables
        integer :: id, jd                                                   ! counters
        integer :: nr_np                                                    ! nr. of next processes found
        integer, allocatable :: next_procs_loc(:)                           ! local next process

        ! initialize
        nr_np = 0
        wl_empty = .true.
        allocate(next_procs_loc(n_procs))

        ! find the next rank
        do id = 1,n_procs-1
            ! wrap around n_procs-1 to zero
            next_procs_loc(nr_np+1) = mod(rank+id,n_procs)

            proc_types: do jd = 1,size(proc_type)
                if (wl(next_procs_loc(nr_np+1)+1).eq.proc_type(jd)) then    ! arrays start at 1, ranks at 0
```

```
                    wl_empty = .false.
                    nr_np = nr_np + 1
                    exit proc_types
                end if
            end do proc_types
        end do

        if (.not.wl_empty .and. present(next_procs)) then
            if (allocated(next_procs)) deallocate(next_procs)
            allocate(next_procs(nr_np))
            next_procs = next_procs_loc(1:nr_np)
        end if
    end function wl_empty

    !> Notifies a rank that they can get the lock.
    !!
    !! The signal sent is the rank + 1.
    !!
    !! \note Based on \cite RossAtomicIO.
    !!
    !! \return ierr
    integer function lock_notify(lock_loc,rec_rank) result(ierr)
        use num_vars, only: rank

        character(*), parameter :: rout_name = 'lock_notify'

        ! input / output
        type(lock_type), intent(in) :: lock_loc                                 !< lock
        integer, intent(in) :: rec_rank                                         !< receiving rank

        ! local variables
#if ldebug
        integer :: istat                                                        ! status
#endif

        ! initialize ierr
        ierr = 0

        call mpi_send(rank+1,1,mpi_integer,rec_rank,lock_loc%wu_tag,&
            &mpi_comm_world,ierr)
        chckerr('Failed to send notification')

#if ldebug
        if (debug_lock) write(*,*,iostat=istat) trim(lock_header(lock_loc)), &
            &'    notified', rec_rank
#endif
    end function lock_notify

    !> Get notified that the rank can  get the lock.
    !!
    !! \note Based on \cite RossAtomicIO.
    !!
    !! \return ierr
    integer function lock_get_notified(lock_loc) result(ierr)
        character(*), parameter :: rout_name = 'lock_get_notified'

        ! input / output
        type(lock_type), intent(in) :: lock_loc                                 !< lock

        ! local variables
        integer :: dum_buf                                                      ! dummy buffer
#if ldebug
        integer :: istat                                                        ! status
#endif

        ! initialize ierr
        ierr = 0

#if ldebug
        if (debug_lock) write(*,*,iostat=istat) trim(lock_header(lock_loc)), &
            &'    but needs to wait for lock'
#endif
        call mpi_recv(dum_buf,1,mpi_integer,mpi_any_source,&
            &lock_loc%wu_tag,mpi_comm_world,mpi_status_ignore,ierr)
        chckerr('Failed to receive notification')
#if ldebug
        if (debug_lock) write(*,*,iostat=istat) trim(lock_header(lock_loc)), &
            &'    got notified by ',dum_buf-1
#endif
    end function lock_get_notified

    !> Adds, removes or sets  to active a rank from the waiting  list for a lock
    !! and returns the lock waiting list:
    !!
    !! Actions:
    !!  - \c wl_action = 0: remove
    !!  - \c wl_action = 1: add
```

```
    !!  - \c wl_action = 2: active
    !!
    !! Or negative equivalents for non-blocking (NB) procs.
    !!
    !! Optionally, the rank(s)  of the process for which to  perform this action
    !! can  be passed.  This is  useful for  doing the  same action  on multiple
    !! processes.
    !!
    !! \note Based on \cite RossAtomicIO.
    !!
    !! \ldebug
    !!
    !! \return ierr
    integer function lock_wl_change(wl_action,blocking,lock_loc,wl,ranks) &
        &result(ierr)
        use num_vars, only: n_procs, rank

        character(*), parameter :: rout_name = 'lock_wl_change'

        ! input / output
        integer, intent(in) :: wl_action                                    !< action to perform
        logical, intent(in) :: blocking                                     !< the ranks to be changed are blocking
        type(lock_type), intent(inout) :: lock_loc                          !< lock
        integer, intent(inout), allocatable :: wl(:)                        !< waiting list
        integer, intent(in), optional :: ranks(:)                           !< rank(s) for which to perform option

        ! local variables
        integer :: id                                                       ! counter
        integer :: n_ranks                                                  ! number of ranks to set
        integer :: put_val                                                  ! value to put
        integer, allocatable :: ranks_loc(:)                                ! local ranks
        integer(kind=MPI_ADDRESS_KIND) :: one = 1                           ! one
        integer(kind=MPI_ADDRESS_KIND) :: disp                              ! local displacement
        integer :: ln                                                       ! local length
#if ldebug
        integer :: istat                                                    ! status
        integer(kind=8) :: window_time(2)                                   ! time that a window is locked
#endif

        ! initialize ierr
        ierr = 0

        ! set value to put
        put_val = wl_action
        if (.not.blocking) put_val = -put_val                               ! NB indicated in waiting list using a negative value
        !allocate(wl_excl(n_procs-1))
        if (allocated(wl)) deallocate(wl)
        allocate(wl(n_procs))

        ! set local ranks if not present
        if (.not.present(ranks)) then
            allocate(ranks_loc(1))
            ranks_loc(1) = rank
        else
            allocate(ranks_loc(size(ranks)))
            ranks_loc = ranks
        end if
        n_ranks = size(ranks_loc)

        ! get window to lock
#if ldebug
        if (debug_lock) call system_clock(window_time(1))
#endif
        call mpi_win_lock(mpi_lock_exclusive,0,0,lock_loc%wl_win,ierr)
        chckerr('Failed to lock window')

        ! get previous list and put value for current ranks
        ! Do this by looking at every rank that is to be put, whether there are
        ! values to be gotten between the previous put and the current put.
        ! Finally, do one more get for the ranks between  the last put  and the
        ! end.
        do id = 1,n_ranks+1
            ! set local displacement and length
            if (id.eq.1) then
                disp = 0
                ln = ranks_loc(1)
            else
                disp = disp + ln + 1                                        ! previous step used ln gets and 1 put
                if (id.lt.n_ranks+1) then
                    ln = ranks_loc(id)-ranks_loc(id-1)-1
                else
                    ln = n_procs-ranks_loc(id-1)-1
                end if
            end if

            ! perform the gets between this rank and previous rank
```

```
            if (ln.gt.0) then                                                   ! if there is something to get between last put and this one
                call mpi_get(wl(int(disp)+1:int(disp)+ln),ln,mpi_integer,0,&
                    &disp,ln,mpi_integer,lock_loc%wl_win,ierr)
                chckerr('Failed to get waiting list')
            end if

            ! perform the single put up to the current rank if not last piece
            if (id.lt.n_ranks+1) then
                call mpi_put(put_val,1,mpi_integer,0,ranks_loc(id)*one,1,&
                    &mpi_integer,lock_loc%wl_win,ierr)
                chckerr('Failed to add to waiting list')
                wl(ranks_loc(id)+1) = put_val
            end if
        end do

        ! unlock window
        call mpi_win_unlock(0,lock_loc%wl_win,ierr)
        chckerr('Failed to lock window')
#if ldebug
        if (debug_lock) call system_clock(window_time(2))
#endif

#if ldebug
        if(debug_lock) write(*,*,iostat=istat) trim(lock_header(lock_loc)),&
            &'    time: '//&
            &trim(r2strt(1.e-9_dp*(window_time(2)-window_time(1)))),&
            &' waiting list:', wl
#endif
    end function lock_wl_change

#if ldebug
    !> Returns the header for lock debug messages.
    !!
    !! \ldebug
    character(len=max_str_ln) function lock_header(lock_loc) result(header)
        use num_vars, only: rank

        ! input / output
        type(lock_type), intent(in) :: lock_loc                                 !< lock

        ! local variables
        integer(kind=8) :: clock                                                ! current clock
        character(len=2) :: block_char                                          ! BL for blocking and NB for non-blocking processes

        ! get clock
        call system_clock(clock)

        ! set block_char
        if (lock_loc%blocking) then
            block_char = 'BL'
        else
            block_char = 'NB'
        end if

        header = trim(ii2str(clock))//' '//trim(i2str(lock_loc%wu_tag))//' '//&
            &block_char//' '//trim(i2str(rank))//'-'
    end function lock_header
#endif
end module mpi_utilities
```

D.54 /Users/toonweyens/Code/PB3D/Modules/MPI_vars.f90 File Reference

Interfaces and Types

- type mpi_vars::lock_type

  lock type

Modules

- module mpi_vars

  Variables pertaining to MPI.

Functions/Subroutines

- integer function, public mpi_vars::init_lock (lock_loc, wu_tag)

  Initializes a lock.
- integer function, public mpi_vars::dealloc_lock (lock_loc)

  Deallocates a lock.

Variables

- type(lock_type), public mpi_vars::hdf5_lock

  HDF5 lock.

D.55 MPI_vars.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------!
!> Variables pertaining to MPI.
!-----------------------------------------------------------------------!
module mpi_vars
#include <PB3D_macros.h>
    use num_vars, only: max_str_ln, dp, plot_dir, data_dir, script_dir
    use str_utilities, only: i2str, r2str, r2strt
    use mpi

    implicit none
    private
    public init_lock, dealloc_lock, &
        &hdf5_lock

    !> lock type
    !!
    !! There is a blocking (BL) and  a nonblocking (NB) version where the former
    !! requires an exclusive lock and the latter  a shared one. This is saved in
    !! the variable \c blocking.
    !!
    !! NB processes  that get the lock  directly on request (meaning  that there
    !! were no  other processes in  the queue) notify  directly all the  next NB
    !! processes after gaining access. It also sets their status to active. When
    !! a NB process gains the lock when notified after waiting, it does not have
    !! to check for other  NB processes, as this has been  done by the notifying
    !! process.
    !!
    !! A BL process retains exclusive access upon receipt of the lock. Similarly
    !! to NB processes, if the receipt was  direct on request, the status is set
    !! to active, but only of this NB process.
    !!
    !! When returning the lock, all BL  processes and NB that find themselves to
    !! be the last  active NB process, scan  the waiting list and  pass the lock
    !! preferably to another BL process to notify. If not available, it searches
    !! for all the NB processes to notify together.
    !!
    !! The advantage of prefering BL processes after finishing a process is that
    !! this  way  NB  processes  are  accumulated,  and  then  quickly  finished
    !! afterwards.
    !!
    !! \note  Every process  in  the  waiting queue  will  eventually receive  a
    !! notification.
    !!
    !! Scheme:
    !! <table>
    !!  <tr> <th> <th> request access <th> gain acces <th> return access
    !!  <tr> <th> BL
    !!      <td> <ul><li> add to queue <li> wait </ul>
    !!      <td> if direct: <ul><li> activate </ul>
    !!      <td> <ul><li> remove from queue <li> find next BL/NB
    !!          <li> notify <li> activate all </ul>
    !!  <tr> <th> NB
    !!      <td> <ul><li> add to queue <li> wait </ul>
    !!      <td> if direct: <ul><li> find next NB </ul>
    !!          for all next NB: <ul> <li> notify <li> activate </ul>
    !!      <td> always: <ul><li> remove from queue </ul>
```

```
    !!          if last NB: <ul><li> find next BL/NB
    !!          <li> notify <li> activate all </ul>
    !! </table>
    !! with preference BL > NB(s).
    !!
    !! \see This is based on an extension of the ideas in \c RossAtomicIO.
    type, public :: lock_type
        integer, allocatable :: wl(:)                                   !< waiting list
        integer :: wl_win                                               !< window to waiting list
        integer :: wu_tag                                               !< wakeup tag
        logical :: blocking                                             !< is a normal blocking process
    contains
        !> initialize
        procedure :: init => init_lock                                  !< initialize
        !> deallocate
        procedure :: dealloc => dealloc_lock                            !< deallocate
    end type lock_type

    ! global variables
    type(lock_type) :: hdf5_lock                                        !< HDF5 lock

contains
    !> Initializes a lock.
    !!
    !! \note
    !!  -# Should be called collectively.
    !!  -# Every lock should have a unique wakeup tag.
    !!
    !! \return ierr
    integer function init_lock(lock_loc,wu_tag) result(ierr)
        use num_vars, only: n_procs, rank

        character(*), parameter :: rout_name = 'init_lock'

        ! input / output
        class(lock_type), intent(inout) :: lock_loc                     !< lock
        integer, intent(in) :: wu_tag                                   !< wakeup tag

        ! local variables
        integer(kind=MPI_ADDRESS_KIND) :: intlb                         ! lower bound of int type
        integer(kind=MPI_ADDRESS_KIND) :: intex                         ! extent of int type

        ! initialize ierr
        ierr = 0

        call mpi_type_get_extent(mpi_integer,intlb,intex,ierr)
        chckerr('Failed to get extent of int')

        if (rank.eq.0) then                                             ! master
            allocate(lock_loc%wl(n_procs))

            lock_loc%wl = 0
            call mpi_win_create(lock_loc%wl,n_procs*intex,int(intex),&
                &mpi_info_null,mpi_comm_world,lock_loc%wl_win,ierr)
            chckerr('Failed to create window to lock_loc')
        else
            allocate(lock_loc%wl(0))

            call mpi_win_create(lock_loc%wl,0*intex,int(intex),mpi_info_null,&
                &mpi_comm_world,lock_loc%wl_win,ierr)
            chckerr('Failed to create window to lock_loc')
        end if

        lock_loc%wu_tag = wu_tag

        ! set fences
        call mpi_win_fence(0,lock_loc%wl_win,ierr)
        chckerr('Couldn''t set fence')
    end function init_lock

    !> Deallocates a lock.
    !!
    !! \note Should be called collectively.
    !!
    !! \return ierr
    integer function dealloc_lock(lock_loc) result(ierr)
        character(*), parameter :: rout_name = 'dealloc_lock'

        ! input / output
        class(lock_type), intent(inout) :: lock_loc                     !< lock

        ! initialize ierr
        ierr = 0

        ! free lock window
        call mpi_win_free(lock_loc%wl_win,ierr)
        chckerr('Failed to free window to HDF5_lock')
```

```

        ! deallocate lock
        deallocate(lock_loc%wl)
    end function dealloc_lock
end module mpi_vars
```

D.56 /Users/toonweyens/Code/PB3D/Modules/num_ops.f90 File Reference

Interfaces and Types

- interface num_ops::calc_zero_hh

  Finds the zero of a function using Householder iteration.

Modules

- module num_ops

  Numerical operations.

Functions/Subroutines

- character(len=max_str_ln) function, public num_ops::calc_zero_zhang (zero, fun, x_int_in)

  Finds the zero of a function using Zhang's method, which is simpler than Brent's method.

Variables

- logical, public num_ops::debug_calc_zero = .false.

  plot debug information for calc_zero

D.57 num_ops.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------!
!> Numerical operations.
!-----------------------------------------------------------------------!
module num_ops
#include <PB3D_macros.h>
    use str_utilities
    use messages
    use num_vars, only: dp, iu, max_str_ln, pi
    use num_utilities
    use output_ops

    implicit none
    private
    public calc_zero_hh, calc_zero_zhang
#if ldebug
    public debug_calc_zero
#endif

    ! global variables
#if ldebug
    !> \ldebug
    logical :: debug_calc_zero = .false.                                !< plot debug information for calc_zero
```

```
#endif

    ! interfaces

    !> \public Finds the zero of a function using Householder iteration.
    !!
    !! If something goes  wrong, by default multiple tries can  be attempted, by
    !! backtracking on the correction by  multiplying it by a relaxation factor.
    !! This can be done \c max_nr_backtracks times.
    !!
    !! If still nothing is achieved, an error message is returned,
    !! that is empty otherwise.
    interface calc_zero_hh
        !> \public
        module procedure calc_zero_hh_0d
        !> \public
        module procedure calc_zero_hh_3d
    end interface

contains
    !> \private 0-D version
    !! \param[inout] fun <tt>fun(x,ord)</tt> with
    !!  - <tt>x</tt> abscissa
    !!  - <tt>ord</tt> order of derivative
    !!  - <tt>fun</tt> ordinate
    function calc_zero_hh_0d(zero,fun,ord,guess,max_nr_backtracks,output) &
        &result(err_msg)
        use num_vars, only: max_it_zero, tol_zero, max_nr_backtracks_hh

        ! input / output
        real(dp), intent(inout) :: zero                                         !< output
        interface
            function fun(x,ord)                                                 ! the function
                use num_vars, only: dp
                real(dp), intent(in) :: x
                integer, intent(in) :: ord
                real(dp) :: fun
            end function fun
        end interface
        integer, intent(in) :: ord                                              !< order of solution
        real(dp), intent(in) :: guess                                           !< first guess
        integer, intent(in), optional :: max_nr_backtracks                      !< max nr. of tries with different relaxation factors
        logical, intent(in), optional :: output                                 !< give output on convergence
        character(len=max_str_ln) :: err_msg                                    !< possible error message

        ! local variables
        integer :: id, jd, kd                                                   ! counters
        integer :: max_nr_backtracks_loc                                        ! local max_nr_backtracks
        logical :: output_loc                                                   ! local output
        logical :: relaxed_enough                                               ! whether step was relaxed enough to get a smaller new value
        real(dp) :: zero_new                                                    ! new zero calculated
        real(dp) :: fun_new                                                     ! function value at new zero
        real(dp) :: corr                                                        ! correction
        real(dp), allocatable :: fun_vals(:)                                    ! function values
#if ldebug
        real(dp), allocatable :: corrs(:)                                       ! corrections for all steps
        real(dp), allocatable :: values(:)                                      ! values for all steps
#endif

        ! initialize error message
        err_msg = ''

        ! set up local output
        output_loc = .false.
        if (present(output)) output_loc = output

        ! set up zero
        zero = guess

        ! set up local max_nr_backtracks
        max_nr_backtracks_loc = max_nr_backtracks_hh
        if (present(max_nr_backtracks)) &
            &max_nr_backtracks_loc = max_nr_backtracks

        ! initialize function values
        allocate(fun_vals(0:ord))

#if ldebug
        if (debug_calc_zero) then
            ! set up corrs
            allocate(corrs(max_it_zero))
            allocate(values(max_it_zero))
        end if
#endif

```

```
        ! loop over iterations
        hh: do jd = 1,max_it_zero
            ! calculate function values
            do kd = 0,ord
                fun_vals(kd) = fun(zero,kd)
            end do

            ! correction to theta_HH
            select case (ord)
                case (1)                                                    ! Newton-Rhapson
                    corr = -fun_vals(0)/fun_vals(1)
                case (2)                                                    ! Halley
                    corr = -fun_vals(0)*fun_vals(1)/&
                        &(fun_vals(1)**2 - 0.5_dp*&
                        &fun_vals(0)*fun_vals(2))
                case (3)                                                    ! 3rd order
                    corr = -(6*fun_vals(0)*fun_vals(1)**2-&
                        &3*fun_vals(0)**2*fun_vals(2))/&
                        &(6*fun_vals(1)**3 - 6*fun_vals(0)*&
                        &fun_vals(1)*fun_vals(2)+&
                        &fun_vals(0)**2*fun_vals(3))
            end select
#if ldebug
            if (debug_calc_zero) then
                corrs(jd) = corr
                values(jd) = zero
            end if
#endif
            relaxed_enough = .false.
            do id = 1,max_nr_backtracks_loc
                zero_new = zero + corr                                      ! propose a new zero

                fun_new = fun(zero_new,0)                                   ! calculate function value there

                if (abs(fun_new)-abs(fun_vals(0)).le.0._dp) then            ! error has decreased
                    relaxed_enough = .true.
                    zero = zero_new
                    exit
                end if
                corr = corr*0.5_dp
            end do

            ! check for relaxation
            if (.not.relaxed_enough) then
                err_msg = trim(i2str(max_nr_backtracks_hh))//' backtracks were &
                    &not sufficient to get a better guess for the zero'
                zero = 0.0_dp
            else
                ! output
                if (output_loc) call writo(trim(i2str(jd))//' / '&
                    &//trim(i2str(max_it_zero))//&
                    &' - relative error: '//&
                    &trim(r2str(abs(corr)))//' (tol: '//&
                    &trim(r2str(tol_zero))//')')

                ! check for convergence
                if (abs(corr).lt.tol_zero) then
#if ldebug
                    if (debug_calc_zero) &
                        &call plot_evolution(corrs(1:jd),values(1:jd))
#endif
                    return
                else if (jd .eq. max_it_zero) then
#if ldebug
                    if (debug_calc_zero) then
                        call plot_evolution(corrs,values)
                        deallocate(corrs)
                        deallocate(values)
                    end if
#endif
                    err_msg = 'Not converged after '//trim(i2str(jd))//&
                        &' iterations, with residual '//&
                        &trim(r2strt(corr))//' and final value '//&
                        &trim(r2strt(zero))
                    zero = 0.0_dp
                end if
            end if
        end do hh
#if ldebug
    contains
        ! plots corrections
        !> \private
        subroutine plot_evolution(corrs,values)
            ! input / output
            real(dp), intent(in) :: corrs(:)                                ! corrections
            real(dp), intent(in) :: values(:)                               ! values

```

```
            ! local variables
            real(dp) :: min_x, max_x                                    ! min. and max. x for which to plot the function
            real(dp) :: extra_x = 1._dp                                 ! how much bigger to take the plot interval than just min and max
            real(dp) :: delta_x                                         ! interval between zero and guess
            integer :: n_x = 200                                        ! how many x values to plot
            real(dp), allocatable :: x_out(:,:)                         ! output x of plot
            real(dp), allocatable :: y_out(:,:)                         ! output y of plot

            ! set up min and max x
            min_x = min(zero,guess)
            max_x = max(zero,guess)
            delta_x = max_x-min_x
            min_x = min_x - delta_x*extra_x
            max_x = max_x + delta_x*extra_x

            ! set up output of plot
            allocate(x_out(n_x,2))
            allocate(y_out(n_x,2))
            x_out(:,1) = [(min_x+(max_x-min_x)*(jd-1._dp)/(n_x-1),jd=1,n_x)]
            x_out(:,2) = [(min_x+(max_x-min_x)*(jd-1._dp)/(n_x-1),jd=1,n_x)]
            y_out(:,1) = [(fun(min_x+(max_x-min_x)*(jd-1._dp)/(n_x-1),0),&
                &jd=1,n_x)]
            y_out(:,2) = [(fun(min_x+(max_x-min_x)*(jd-1._dp)/(n_x-1),1),&
                &jd=1,n_x)]

            ! plot function
            call writo('Initial guess: '//trim(r2str(guess)))
            call writo('Resulting zero: '//trim(r2str(zero)))
            call print_ex_2d(['function  ','derivative'],'',y_out,x=x_out)

            ! user output
            call writo('Last correction '//trim(r2strt(corrs(size(corrs))))&
                &//' with tolerance '//trim(r2strt(tol_zero)))

            ! plot corrections and values
            call print_ex_2d('corrs','',corrs)
            call print_ex_2d('values','',values)
        end subroutine plot_evolution
#endif
    end function calc_zero_hh_0d
    !> \private 3-D version
    !! \param[inout] fun <tt>fun(dims,x,ord)</tt> with
    !!  - <tt>dims(3)</tt> dimension of abscissa and ordinate
    !!  - <tt>x(dims(1),dims(2),dims(3))</tt> abscissa
    !!  - <tt>ord</tt> order of derivative
    !!  - <tt>fun(dims(1),dims(2),dims(3))</tt> ordinate
    function calc_zero_hh_3d(dims,zero,fun,ord,guess,max_nr_backtracks,output) &
        &result(err_msg)
        use num_vars, only: max_it_zero, tol_zero, max_nr_backtracks_hh

        ! input / output
        integer, intent(in) :: dims(3)                                  !< dimensions of the problem
        real(dp), intent(inout) :: zero(dims(1),dims(2),dims(3))        !< output
        interface
            function fun(dims,x,ord)                                    ! the function
                use num_vars, only: dp
                integer, intent(in) :: dims(3)
                real(dp), intent(in) :: x(dims(1),dims(2),dims(3))
                integer, intent(in) :: ord
                real(dp) :: fun(dims(1),dims(2),dims(3))
            end function fun
        end interface
        integer, intent(in) :: ord                                      !< order of solution
        real(dp), intent(in) :: guess(dims(1),dims(2),dims(3))          !< first guess
        integer, intent(in), optional :: max_nr_backtracks              !< max nr. of backtracks
        logical, intent(in), optional :: output                         !< give output on convergence
        character(len=max_str_ln) :: err_msg                            !< possible error message

        ! local variables
        integer :: id, jd, kd                                           ! counters
        integer :: max_nr_backtracks_loc                                ! local max_nr_backtracks
        integer :: mc_ind(3)                                            ! index of maximum correction
        logical :: output_loc                                           ! local output
        logical :: relaxed_enough                                       ! relaxed enough
        real(dp) :: zero_new(dims(1),dims(2),dims(3))                   ! new zeros calculated
        real(dp) :: fun_new(dims(1),dims(2),dims(3))                    ! function values at new zero
        real(dp) :: corr(dims(1),dims(2),dims(3))                       ! correction
        real(dp), allocatable :: fun_vals(:,:,:,:)                      ! function values
#if ldebug
        real(dp), allocatable :: corrs(:,:,:,:)                         ! corrections for all steps
        real(dp), allocatable :: values(:,:,:,:)                        ! values for all steps
#endif

        ! initialize error message
        err_msg = ''
```

```fortran

        ! test
        if (ord.lt.1 .or. ord.gt.3) then
            zero = 0.0_dp
            err_msg = 'only orders 1 (Newton-Rhapson), 2 (Halley) and 3 &
                &implemented, not '//trim(i2str(ord))
            return
        end if

        ! set up local output
        output_loc = .false.
        if (present(output)) output_loc = output

        ! set up zero
        zero = guess

        ! set up local max_nr_backtracks
        max_nr_backtracks_loc = max_nr_backtracks_hh
        if (present(max_nr_backtracks)) &
            &max_nr_backtracks_loc = max_nr_backtracks

        ! initialize function values
        allocate(fun_vals(dims(1),dims(2),dims(3),0:ord))

#if ldebug
        if (debug_calc_zero) then
            ! set up corrs
            allocate(corrs(dims(1),dims(2),dims(3),max_it_zero))
            allocate(values(dims(1),dims(2),dims(3),max_it_zero))
        end if
#endif

        ! loop over iterations
        hh: do jd = 1,max_it_zero
            ! calculate function values
            do kd = 0,ord
                fun_vals(:,:,:,kd) = fun(dims,zero,kd)
            end do

            ! correction to theta_HH
            select case (ord)
                case (1)                                                ! Newton-Rhapson
                    corr = -fun_vals(:,:,:,0)/fun_vals(:,:,:,1)
                case (2)                                                ! Halley
                    corr = -fun_vals(:,:,:,0)*fun_vals(:,:,:,1)/&
                        &(fun_vals(:,:,:,1)**2 - 0.5_dp*&
                        &fun_vals(:,:,:,0)*fun_vals(:,:,:,2))
                case (3)                                                ! 3rd order
                    corr = -(6*fun_vals(:,:,:,0)*fun_vals(:,:,:,1)**2-&
                        &3*fun_vals(:,:,:,0)**2*fun_vals(:,:,:,2))/&
                        &(6*fun_vals(:,:,:,1)**3 - 6*fun_vals(:,:,:,0)*&
                        &fun_vals(:,:,:,1)*fun_vals(:,:,:,2)+&
                        &fun_vals(:,:,:,0)**2*fun_vals(:,:,:,3))
            end select
#if ldebug
            if (debug_calc_zero) then
                corrs(:,:,:,jd) = corr
                values(:,:,:,jd) = zero
            end if
#endif
            relaxed_enough = .false.
            do id = 1,max_nr_backtracks_loc
                zero_new = zero                                         ! otherwise some values might stay unitialized
                where (abs(corr).gt.tol_zero) zero_new = zero + corr    ! propose a new zero

                fun_new = fun(dims,zero_new,0)                          ! calculate function value there

#if ldebug
                if (debug_calc_zero) &
                    &call writo('backtrack '//trim(i2str(id))//' of max. '//&
                    &trim(i2str(max_nr_backtracks_loc))//' - criterion: '//&
                    &trim(r2str(maxval(abs(fun_new)-abs(fun_vals(:,:,:,0))))))//&
                    &' <= 0?')
#endif
                if (maxval(abs(fun_new)-abs(fun_vals(:,:,:,0))).le.0._dp) then  ! error has decreased
                    relaxed_enough = .true.
                    zero = zero_new
                    exit
                end if
                corr = corr*0.5_dp
            end do

            ! check for relaxation
            if (.not.relaxed_enough) then
                err_msg = trim(i2str(max_nr_backtracks_hh))//' backtracks were &
                    &not sufficient to get a better guess for the zero'
```

```
                zero = 0.0_dp
                exit
            else
                ! output
                if (output_loc) call writo(trim(i2str(jd))//' / '&
                    &//trim(i2str(max_it_zero))//&
                    &' - maximum relative error: '//&
                    &trim(r2str(maxval(abs(corr))))//' (tol: '//&
                    &trim(r2str(tol_zero))//')')

                ! check for convergence
                if (maxval(abs(corr)).lt.tol_zero) then
#if ldebug
                    if (debug_calc_zero) call &
                        &plot_evolution(corrs(:,:,:,1:jd),values(:,:,:,1:jd))
#endif
                    return
                else if (jd .eq. max_it_zero) then
#if ldebug
                    if (debug_calc_zero) then
                        call plot_evolution(corrs,values)
                        deallocate(corrs)
                        deallocate(values)
                    end if
#endif
                    mc_ind = maxloc(abs(corr))
                    err_msg = 'Not converged after '//trim(i2str(jd))//&
                        &' iterations, with maximum residual '//&
                        &trim(r2strt(corr(mc_ind(1),mc_ind(2),mc_ind(3))))&
                        &//' and final value '//trim(r2strt(&
                        &zero(mc_ind(1),mc_ind(2),mc_ind(3))))
                    zero = 0.0_dp
                end if
            end if
        end do hh
#if ldebug
    contains
        ! plots corrections
        !> \private
        subroutine plot_evolution(corrs,values)
            ! input / output
            real(dp), intent(in) :: corrs(:,:,:,:)                          ! corrections
            real(dp), intent(in) :: values(:,:,:,:)                         ! values

            ! local variables
            integer :: n_corrs                                              ! number of corrections
            character(len=max_str_ln) :: var_name(1)                        ! names of variable
            character(len=max_str_ln) :: file_name                          ! name of file

            ! initialize
            n_corrs = size(corrs,4)
            mc_ind = maxloc(abs(corrs(:,:,:,n_corrs)))

            ! user output
            call writo('Last maximum correction '//&
                &trim(r2strt(corrs(mc_ind(1),mc_ind(2),mc_ind(3),n_corrs)))&
                &//' and final value '//trim(r2strt(&
                &values(mc_ind(1),mc_ind(2),mc_ind(3),n_corrs)))//&
                &' with tolerance '//trim(r2strt(tol_zero)))

            ! plot corrs
            file_name = 'corrs'
            var_name = 'var'

            call plot_hdf5(var_name,file_name,corrs,col=2,&
                &descr='corrections')

            ! plot values
            file_name = 'values'
            var_name = 'var'

            call plot_hdf5(var_name,file_name,values,col=2,&
                &descr='values')
        end subroutine plot_evolution
#endif
    end function calc_zero_hh_3d

    !> Finds the zero of a function  using Zhang's method, which is simpler than
    !! Brent's method.
    !!
    !! Taken  from  from  Steven  Stage's  correction  of  Zhang's  paper
    !! \cite zhang2011improvement.
    !!
    !! Unlike Householder, Zhang's method needs  an interval \c x_int_in to work
    !! in,  not  a guess.  Also,  it  does not  require  the  derivative of  the
    !! function.
    !!
```

```
    !! The routine returns  an error message if  no zero is found,  and which is
    !! empty otherwise.
    !!
    !! \note The interval \c x_int_in needs to be so that the function values at
    !! either end are of different value.
    !! \param[inout] fun <tt>fun(x)</tt> with
    !!  - <tt>x</tt> abscissa
    !!  - <tt>fun</tt> ordinate
    function calc_zero_zhang(zero,fun,x_int_in) result(err_msg)
        use num_vars, only: max_it_zero, tol_zero

        ! input / output
        real(dp), intent(inout) :: zero                                     !< output
        interface
            function fun(x)                                                 ! the function
                use num_vars, only: dp
                real(dp), intent(in) :: x
                real(dp) :: fun
            end function fun
        end interface
        real(dp), intent(in) :: x_int_in(2)                                 !< interval
        character(len=max_str_ln) :: err_msg                                !< possible error message

        ! local variables
        logical :: converged                                                ! whether converged
        integer :: jd                                                       ! counter
        integer :: id_loc                                                   ! local id (either 1 or 2)
        real(dp) :: x_int(2)                                                ! points of current interval [a,b]
        real(dp) :: x_mid                                                   ! x at midpoint [c]
        real(dp) :: x_rule                                                  ! x found with rule [s]
        real(dp) :: x_temp                                                  ! temporary x
        real(dp) :: fun_int(2)                                              ! function at x_int
        real(dp) :: fun_mid                                                 ! function at x_mid
        real(dp) :: fun_rule                                                ! function at x_rule
#if ldebug
        real(dp), allocatable :: x_ints(:,:)                                ! bounds for all steps
#endif

        ! initialize error message and zero
        err_msg = ''
        zero = 0.0_dp

        ! initialize from input
        x_int(1) = minval(x_int_in)
        x_int(2) = maxval(x_int_in)
        fun_int = [fun(x_int(1)),fun(x_int(2))]

        ! test whether already zero
        if (abs(fun_int(1)).lt.tol_zero) then
            zero = x_int(1)
            return
        end if
        if (abs(fun_int(2)).lt.tol_zero) then
            zero = x_int(2)
            return
        end if

        ! test whether different sign
        if (product(fun_int).gt.0) then
            err_msg = 'Function values on starting interval have same sign'
            return
        end if

        ! intialize converged
        converged = .false.

#if ldebug
        if (debug_calc_zero) then
            ! set up x_ints
            allocate(x_ints(max_it_zero,2))
        end if
#endif

        ! iterations
        do jd = 1,max_it_zero
            ! calculate midpoint and function value
            x_mid = 0.5_dp*sum(x_int)
            fun_mid = fun(x_mid)

            if (abs(fun_int(1)-fun_mid).gt.tol_zero .and. &
                &abs(fun_int(2)-fun_mid).gt.tol_zero) then
                ! three  different  function   values:   use  inverse  quadratic
                ! interpolation
                x_rule = fun_mid*fun_int(2)*x_int(1)/&
                    &((fun_int(1)-fun_mid)*(fun_int(1)-fun_int(2))) + &
                    &fun_int(1)*fun_int(2)*x_mid/&
                    &((fun_mid-fun_int(1))*(fun_mid-fun_int(2))) + &
```

```
                    &fun_int(1)*fun_mid*x_int(2)/&
                    &((fun_int(2)-fun_mid)*(fun_int(2)-fun_mid))

                if (x_int(1).lt.x_rule .and. x_rule.lt.x_int(2)) then
                    ! found x within interval
                else
                    ! use bisection instead
                    if (fun_int(1)*fun_mid.lt.0) then                          ! [a,c] contains sign change
                        id_loc = 1
                    else if (fun_int(2)*fun_mid.lt.0) then                     ! [b,c] contains sign change
                        id_loc = 2
                    else
                        err_msg = 'Something went wrong...'
                        converged = .true.
                    end if
                    if (.not.converged) x_rule = 0.5_dp*(x_int(id_loc)+x_mid)
                end if
            else
                ! two function values overlap so use secant rule
                if (fun_int(1)*fun_mid.lt.0) then                              ! [a,c] contains sign change
                    id_loc = 1
                else if (fun_int(2)*fun_mid.lt.0) then                         ! [b,c] contains sign change
                    id_loc = 2
                else
                    err_msg = 'Something went wrong...'
                    converged = .true.
                end if
                if (.not.converged) x_rule = x_int(id_loc) - &
                    &fun_int(id_loc)*(x_int(id_loc)-x_mid)/&
                    &(fun_int(id_loc)-fun_mid)
            end if

            ! calculate fun(s)
            fun_rule = fun(x_rule)

            ! make sure x_mid < x_rule
            if (x_mid.gt.x_rule) then
                x_temp = x_rule
                x_rule = x_mid
                x_mid = x_temp
            end if

            ! check whether [c,s] contains root
            if (fun_mid*fun_rule.lt.0) then
                x_int(1) = x_mid
                x_int(2) = x_rule
            else
                if (fun_int(1)*fun_mid.lt.0) then
                    x_int(2) = x_mid
                else
                    x_int(1) = x_rule
                end if
            end if

#if ldebug
            if (debug_calc_zero) then
                x_ints(jd,:) = x_int
            end if
#endif

            ! check for convergence
            fun_int = [fun(x_int(1)),fun(x_int(2))]
            if (abs(fun_int(1)).lt.tol_zero) then
                zero = x_int(1)
                converged = .true.
            end if
            if (abs(fun_int(2)).lt.tol_zero) then
                zero = x_int(2)
                converged = .true.
            end if
            if ((x_int(2)-x_int(1)).lt.tol_zero) then
                zero = 0.5_dp*sum(x_int)
                converged = .true.
            end if

            if (converged) then
#if ldebug
                if (debug_calc_zero) call plot_evolution(x_ints(1:jd,:))
#endif
                return
            end if
        end do
#if ldebug
    contains
        ! plots corrections
        !> \private
        subroutine plot_evolution(x_ints)
```

```
            ! input / output
            real(dp), intent(in) :: x_ints(:,:)                                 ! x_intervals

            ! local variables
            real(dp) :: min_x, max_x                                            ! min. and max. x for which to plot the function
            integer :: n_x = 200                                                ! how many x values to plot
            integer :: n_ints                                                   ! nr. of intervals
            real(dp), allocatable :: x_out(:)                                   ! output x of plot
            real(dp), allocatable :: y_out(:)                                   ! output y of plot

            ! set up nr. of intervals
            n_ints = size(x_ints,1)

            ! set up min and max x
            min_x = x_int_in(1)
            max_x = x_int_in(2)

            ! set up output of plot
            allocate(x_out(n_x))
            allocate(y_out(n_x))
            x_out = [(min_x+(max_x-min_x)*(jd-1._dp)/(n_x-1),jd=1,n_x)]
            y_out = [(fun(min_x+(max_x-min_x)*(jd-1._dp)/(n_x-1)),jd=1,n_x)]

            ! plot function
            call writo('Initial interval: ['//trim(r2str(x_int_in(1)))//&
                &'..'//trim(r2str(x_int_in(2)))//']')
            call writo('Resulting zero: '//trim(r2str(zero)))
            call print_ex_2d('function','',y_out,x=x_out)

            ! user output
            call writo('Final interval ['//trim(r2str(x_ints(n_ints,1)))//&
                &'..'//trim(r2str(x_ints(n_ints,2)))//'] with tolerance '//&
                &trim(r2strt(tol_zero)))

            ! plot intervals
            call print_ex_2d(['x_intervals'],'',x_ints)
        end subroutine plot_evolution
#endif
    end function calc_zero_zhang
end module num_ops
```

D.58 /Users/toonweyens/Code/PB3D/Modules/num_utilities.f90 File Reference

Interfaces and Types

- interface num_utilities::add_arr_mult

  Add to an array (3) the product of arrays (1) and (2).
- interface num_utilities::calc_det

  Calculate determinant of a matrix.
- interface num_utilities::calc_inv

  Calculate inverse of square matrix A.
- interface num_utilities::calc_mult

  Calculate matrix multiplication of two square matrices $\overline{AB} = \overline{A}\,\overline{B}$.
- interface num_utilities::conv_mat

  Converts a (symmetric) matrix A with the storage convention described in eq_vars.eq_2_type.
- interface num_utilities::calc_int

  Integrates a function using the trapezoidal rule.
- interface num_utilities::round_with_tol

  Rounds an arry of values to limits, with a tolerance 10^{-5} *that can optionally be modified.*
- interface num_utilities::con2dis

  Convert between points from a continuous grid to a discrete grid.
- interface num_utilities::dis2con

  Convert between points from a discrete grid to a continuous grid.

- interface num_utilities::con

  Either takes the complex conjugate of a square matrix element A, defined on a 3-D grid, or not.
- interface num_utilities::bubble_sort

  Sorting with the bubble sort routine.
- interface num_utilities::order_per_fun

  Order a periodic function to include $0 \ldots 2\pi$ and an overlap.
- interface num_utilities::spline

  Wrapper to the pspline library, making it easier to use for 1-D applications where speed is not the main priority. If spline representations are to be reused, manually use the library.

Modules

- module num_utilities

  Numerical utilities.

Functions/Subroutines

- integer function setup_lim_vals (xb, lim_vals)

  /private set up limit values
- subroutine, public num_utilities::calc_aux_utilities (n_mod)

  Initialize utilities for fast future reference, depending on program style.
- integer function, dimension(:,:), allocatable, public num_utilities::derivs (order, dims)

  Returns derivatives of certain order.
- integer function, public num_utilities::check_deriv (deriv, max_deriv, sr_name)

  checks whether the derivatives requested for a certain subroutine are valid
- integer function, public num_utilities::calc_ext_var (ext_var, var, var_points, ext_point, deriv_in)

  Extrapolates a function.
- integer function, public num_utilities::calc_coeff_fin_diff (deriv, nr, ind, coeff)

  Calculate the coefficients for finite differences.
- integer function, public num_utilities::c (ij, sym, n, lim_n)

  Convert 2-D coordinates (i,j) to the storage convention used in matrices.
- recursive integer function, public num_utilities::fac (n)

  Calculate factorial.
- integer function, public num_utilities::is_sym (n, nn, sym)

  Determines whether a matrix making use of the storage convention in eq_vars.eq_2_type is symmetric or not.
- recursive integer function, public num_utilities::lcm (u, v)

  Returns common multiple using the Euclid's algorithm.
- recursive integer function, public num_utilities::gcd (u, v)

  Returns least denominator using the GCD.
- subroutine, public num_utilities::shift_f (al, bl, cl, a, b, c)

  Calculate multiplication through shifting of fourier modes A and B into C.
- subroutine, public num_utilities::solve_vand (n, a, b, x, transp)

  Solve a Vandermonde system $\overline{A}\,\vec{X} = \vec{B}$.
- integer function, public num_utilities::calc_d2_smooth (fil_n, x, y, d2y, style)

  Calculate second derivative with smoothing formula by Holoborodko, [9].

Variables

- integer, dimension(:,:,:), allocatable, public num_utilities::d

  1-D array indices of derivatives
- integer, dimension(:,:), allocatable, public num_utilities::m

  1-D array indices of metric indices
- integer, dimension(:,:), allocatable, public num_utilities::f

  1-D array indices of Fourier mode combination indices
- logical, public num_utilities::debug_con2dis_reg = .false.

  plot debug information for con2dis_reg()
- logical, public num_utilities::debug_calc_coeff_fin_diff = .false.

  plot debug information for calc_coeff_fin_diff()

D.58.1 Function/Subroutine Documentation

D.58.1.1 setup_lim_vals()

```
integer function spline_real::setup_lim_vals (
            real(dp), intent(in) xb,
            real(dp), dimension(0:3), intent(out) lim_vals)
```

/private set up limit values

Makes use of f_spl

Definition at line 1937 of file num_utilities.f90.

Here is the caller graph for this function:

D.59 num_utilities.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Numerical utilities.
!-----------------------------------------------------------------------------!
module num_utilities
#include <PB3D_macros.h>
#include <wrappers.h>
    use str_utilities
    use messages
    use num_vars, only: dp, iu, max_str_ln, pi

    implicit none
    private
    public calc_ext_var, calc_det, calc_int, add_arr_mult, c, &
        &check_deriv, calc_inv, calc_mult, calc_aux_utilities, derivs, &
        &con2dis, dis2con, round_with_tol, conv_mat, is_sym, con, &
        &calc_coeff_fin_diff, fac, d, m, f, bubble_sort, gcd, lcm, &
        &order_per_fun, shift_f, spline, solve_vand
#if ldebug
    public debug_con2dis_reg, debug_calc_coeff_fin_diff, calc_d2_smooth
#endif

    ! global variables
    integer, allocatable :: d(:,:,:)                                        !< 1-D array indices of derivatives
    integer, allocatable :: m(:,:)                                          !< 1-D array indices of metric indices
    integer, allocatable :: f(:,:)                                          !< 1-D array indices of Fourier mode combination indices
#if ldebug
    logical :: debug_con2dis_reg = .false.                                  !< plot debug information for con2dis_reg() \ldebug
    logical :: debug_calc_coeff_fin_diff = .false.                          !< plot debug information for calc_coeff_fin_diff() \ldebug
#endif

    ! interfaces

    !> \public Add to an  array (3) the product  of arrays (1) and  (2).
    !!
    !! The derivatives  are distributed  between both  acording to  the binomial
    !! theorem. Both arrays are given on a 3-D or 1-D grid.
    !!
    !! \return ierr
    interface add_arr_mult
        !> \public
        module procedure add_arr_mult_3_3
        !> \public
        module procedure add_arr_mult_3_1
        !> \public
        module procedure add_arr_mult_1_1
    end interface

    !> \public Calculate determinant of a matrix
    !!
    !! This  matrix can  be  defined on  a  3-D grid  or  constant. The  storage
    !! convention described in eq_vars.eq_2_type is used.
    !!
    !! In the former  case the size of  the matrix (last two  indices) should be
    !! small,  as the  direct  formula  employing cofactors  is  used through  a
    !! recursive formulation.
    !!
    !! In the latter case, lapack routines are used.
    !!
    !! \see Adapted from
    !! <http://dualm.wordpress.com/2012/01/06/computing-determinant-in-fortran/>
    !!
    !! \return ierr
    interface calc_det
        !> \public
        module procedure calc_det_0d
        !> \public
        module procedure calc_det_3d
    end interface

    !> \public  Calculate inverse of square matrix \c A.
    !!
    !! This  matrix can  be  defined on  a  3-D grid  or  constant. The  storage
    !! convention described in eq_vars.eq_2_type is used.
    !!
    !! In the former  case the size of  the matrix (last two  indices) should be
    !! small, as direct inversion is performed using Cramer's rule.
    !!
    !! In the latter case, lapack routines are used.
```

```
    !!
    !! \return ierr
    interface calc_inv
        !> \public
        module procedure calc_inv_0d
        !> \public
        module procedure calc_inv_3d
    end interface

    !> \public Calculate matrix multiplication of two square matrices
    !! \f$\overline{\text{AB}} = \overline{\text{A}} \ \overline{\text{B}}\f$.
    !!
    !! This  matrix can  be  defined on  a  3-D grid  or  constant. The  storage
    !! convention described in eq_vars.eq_2_type is used.
    !!
    !! \return ierr
    interface calc_mult
        !> \public
        module procedure calc_mult_0d_real
        !> \public
        module procedure calc_mult_3d_real
        !> \public
        module procedure calc_mult_3d_complex
    end interface

    !> \public  Converts a  (symmetric)  matrix A  with  the storage  convention
    !! described in eq_vars.eq_2_type.
    !!
    !! This matric can have elements depending on a 3-D grid or be constant.
    !!
    !!  - If  the matrix  is stored  with n^2 numbers,  only the  lower diagonal
    !!  elements are kept in matrix B.
    !!  - If only the lower diagonal elements are stored, they are copied to the
    !!  upper diagonal ones of the matrix B as well.
    !!
    !! Optionally, the transpose can be calculated.
    !!
    !! \note
    !!  -# The routine does not check whether the matrix is indeed symmetric and
    !!  that information may thus be lost after conversion.
    !!  -#  This routine  makes a  copy of  A so  that by  providing A  as input
    !!  argument for both A and B overwrites A.
    !!
    !! \return ierr
    interface conv_mat
        !> \public
        module procedure conv_mat_3d
        !> \public
        module procedure conv_mat_3d_complex
        !> \public
        module procedure conv_mat_0d
        !> \public
        module procedure conv_mat_0d_complex
    end interface

    !> \public Integrates a function using the trapezoidal rule.
    !!
    !! This function can be defined on an equidistant grid or a regular one:
    !!  - For an equidistant grid,
    !!      \f[\int_1^n f(x) dx = \sum_{k=1}^{n-1}
    !!          {\left(f(k+1)+f(k)\right) \frac{\Delta_x}{2}},\f]
    !!  is used,  with \f$n\f$  the number  of points, which  are assumed  to be
    !!  equidistant with a given step size \f$Delta_x\f$.
    !!  - For a regular grid,
    !!      \f[\int_1^n f(x) dx = \sum_{k=1}^{n-1}
    !!          {\left(f(k+1)+f(k)\right) \frac{x(k+1)-x(k)}{2}}, \f]
    !!  is used, with \f$n\f$ the number of points.
    !!      - Therefore, \f$n\f$  points have to be specified as well as
    !!      \f$n\f$ values  for the function  to be integrated.
    !!      - They  have to be  given in
    !!      ascending order but the step size does not have to be constant.
    !!      - This yields the following difference formula:
    !!      \f[\int_1^n f(x) dx = \int_1^{n-1} f(x) dx +
    !!          \left(f(n)+f(n-1)\right) \frac{x(n)-x(n-1)}{2},\f]
    !!      which is used here
    !!
    !! \note For periodic function, the trapezoidal  rule works well only if the
    !! last point of the grid is included,  i.e. the point where the function is
    !! equal to the first point.
    !!
    !! \return ierr
    interface calc_int
        !> \public
        module procedure calc_int_eqd
        !> \public
        module procedure calc_int_reg
    end interface
```

```

    !>  \public  Rounds  an   arry  of  values  to  limits,   with  a  tolerance
    !! \f$10^{-5}\f$ that can optionally be modified.
    interface round_with_tol
        !> \public
        module procedure round_with_tol_ind
        !> \public
        module procedure round_with_tol_arr
    end interface

    !> \public Convert between points from a continuous grid to a discrete grid.
    !!
    !! This is done by providing either the the limits on the grid (\c lim_c and
    !! \c lim_d), in  which case the grid  is assumed to be  equidistant, or the
    !! grid values themselves, in which case the grid just has to be regular.
    !!
    !! The output is a real value where  the floored integer is the index in the
    !! discrete grid and  the remainder corresponds to the  fraction towards the
    !! next index.  If no solution is  found, a negative value  is outputted, as
    !! well as a message.
    !!
    !! \return ierr
    interface con2dis
        !> \public
        module procedure con2dis_eqd
        !> \public
        module procedure con2dis_reg
    end interface

    !> \public Convert between points from a discrete grid to a continuous grid.
    !!
    !! This is done by providing either the the limits on the grid (\c lim_c and
    !! \c lim_d), in  which case the grid  is assumed to be  equidistant, or the
    !! grid values themselves, in which case the grid just has to be regular.
    !!
    !! The output is a real value. If the discrete value lies outside the range,
    !! in the case of a regular grid,  a negative value is outputted, as well as
    !! a message.
    !!
    !! \return ierr
    interface dis2con
        !> \public
        module procedure dis2con_eqd
        !> \public
        module procedure dis2con_reg
    end interface

    !> \public Either takes the complex conjugate  of a square matrix element A,
    !! defined on a 3-D grid, or not.
    !!
    !! This  is done  depending on  whether the  indices of  the matrix  element
    !! correspond to  the upper (conjugate)  or lower (no  conjugate) triangular
    !! part.
    !!
    !! \note Only for symmetric matrices does this have to be applied.
    interface con
        !> \public
        module procedure con_3d
        !> \public
        module procedure con_2d
        !> \public
        module procedure con_1d
        !> \public
        module procedure con_0d
    end interface

    !> \public Sorting with the bubble sort routine.
    !!
    !! Optionally, the pivots can be given back.
    !!
    !! \note Adapted from <http://rosettacode.org/wiki/Category:Fortran>
    interface bubble_sort
        !> \public
        module procedure bubble_sort_int
        !> \public
        module procedure bubble_sort_real
    end interface

    !> \public Order  a periodic function  to include \f$0\ldots 2\pi\f$  and an
    !! overlap.
    !!
    !! \return ierr
    interface order_per_fun
        !> \public
        module procedure order_per_fun_1
        !> \public
        module procedure order_per_fun_2
```

```
    end interface

    !> \public  Wrapper to  the pspline  library, making  it easier  to use  for
    !! 1-D  applications  where  speed  is  not the  main  priority.  If  spline
    !! representations are to be reused, manually use the library.
    !!
    !! Order  1 (linear),  2 (akima  hermite)  or 3  (cubic) possible.  Boundary
    !! conditions are possible:
    !!  - -1: periodic
    !!  -  0: not-a-knot
    !!  -  1: prescribe first derivative
    !!  -  2: prescribe second derivative
    !!
    !! However, for order 2 boundary condition  2 is not available and for order
    !! 1 none of them.
    !!
    !! Furthermore, derivatives can be specified:
    !!  - up to 1 for order 1 and 2
    !!  - up to 2 for order 3
    !!
    !! Finally, extrapolation can be performed as well.
    !!
    !! \return ierr
    interface spline
        !> \public
        module procedure spline_real
        !> \public
        module procedure spline_complex
    end interface

contains
    !> \private regular version
    integer function calc_int_reg(var,x,var_int) result(ierr)
        character(*), parameter :: rout_name = 'calc_int_reg'

        ! input / output
        real(dp), intent(inout) :: var_int(:)                                   !< integrated variable
        real(dp), intent(in) :: var(:)                                          !< variable to be integrated
        real(dp), intent(in) :: x(:)                                            !< abscissa

        ! local variables
        integer :: n_max
        integer :: kd
        character(len=max_str_ln) :: err_msg                                    ! error message

        ! initialize ierr
        ierr = 0

        ! set n_max
        n_max = size(var)

        ! tests
        if (size(x).ne.n_max) then
            err_msg = 'The arrays of points and values are not of the same &
                &length'
            ierr = 1
            chckerr(err_msg)
        else if (n_max.lt.2) then
            err_msg = 'Asking to integrate with '//trim(i2str(n_max))&
                &//' points. Need at least 2'
            ierr = 1
            chckerr(err_msg)
        end if

        ! calculate integral for all points
        var_int = 0.0_dp

        do kd = 2, n_max
            var_int(kd) = var_int(kd-1) + &
                &(var(kd)+var(kd-1))/2 * (x(kd)-x(kd-1))
        end do
    end function calc_int_reg
    !> \private equidistant version
    integer function calc_int_eqd(var,step_size,var_int) result(ierr)
        character(*), parameter :: rout_name = 'calc_int_eqd'

        ! input / output
        real(dp), intent(inout) :: var_int(:)                                   !< integrated variable
        real(dp), intent(in) :: var(:)                                          !< variable to be integrated
        real(dp), intent(in) :: step_size                                       !< step size of abscissa

        ! local variables
        integer :: n_max
        integer :: kd
        character(len=max_str_ln) :: err_msg                                    ! error message

        ! initialize ierr
```

```
        ierr = 0

        ! set n_max
        n_max = size(var)

        ! tests
        if (n_max.lt.2) then
            err_msg = 'Asking to integrate with '//trim(i2str(n_max))&
                &//' points. Need at least 2'
            ierr = 1
            chckerr(err_msg)
        end if

        ! calculate integral for all points
        var_int = 0.0_dp

        do kd = 2, n_max
            var_int(kd) = var_int(kd-1) + (var(kd)+var(kd-1))/2 * step_size
        end do
    end function calc_int_eqd

    !> \private version with \c arr_1 and \c arr_2 in 3 coords.
    integer function add_arr_mult_3_3(arr_1,arr_2,arr_3,deriv) result(ierr)
        character(*), parameter :: rout_name = 'add_arr_mult_3_3'

        ! input / output
        real(dp), intent(in) :: arr_1(1:,1:,1:,0:,0:,0:)                        !< \c arr_1
        real(dp), intent(in) :: arr_2(1:,1:,1:,0:,0:,0:)                        !< \c arr_2
        real(dp), intent(out) :: arr_3(1:,1:,1:)                                !< \c arr_3
        integer, intent(in) :: deriv(3)                                         !< derivatives

        ! local variables
        real(dp) :: bin_fac(3)                                                  ! binomial factor for norm., pol. and tor. sum
        integer :: r, m, n                                                      ! current degree of norm., pol., tor. derivatives
        integer :: kd                                                           ! normal counter
        character(len=max_str_ln) :: err_msg                                    ! error message

        ! initialize ierr
        ierr = 0

        ! tests
        if (size(arr_1,1).ne.size(arr_2,1) .or. size(arr_1,2).ne.size(arr_2,2) &
            &.or. size(arr_1,3).ne.size(arr_2,3)) then
            err_msg = 'Arrays 1 and 2 need to have the same size'
            ierr = 1
            chckerr(err_msg)
        end if
        if (size(arr_1,1).ne.size(arr_3,1) .or. size(arr_1,2).ne.size(arr_3,2) &
            &.or. size(arr_1,3).ne.size(arr_3,3)) then
            err_msg = 'Arrays 1 and 2 need to have the same size as the &
                &resulting array 3'
            ierr = 1
            chckerr(err_msg)
        end if
        do kd = 1,3
            if (size(arr_1,3+kd).lt.deriv(kd)+1 .or. &
                &size(arr_2,3+kd).lt.deriv(kd)+1) then
                err_msg = 'Arrays 1 and 2 do not provide deriv. orders for a &
                    &derivative of order ['//trim(i2str(deriv(1)))//','//&
                    &trim(i2str(deriv(2)))//','//trim(i2str(deriv(3)))//'&
                    &] in coordinate '//trim(i2str(kd))
                ierr = 1
                chckerr(err_msg)
            end if
        end do

        ! distribute normal and angular derivatives using binomial theorem
        ! normal derivatives
        do r = 0,deriv(1)
            if (r.eq.0) then                                                    ! first term in sum
                bin_fac(1) = 1.0_dp
            else
                bin_fac(1) = bin_fac(1)*(deriv(1)-(r-1))/r
            end if
            ! poloidal derivatives
            do m = 0,deriv(2)
                if (m.eq.0) then                                                ! first term in sum
                    bin_fac(2) = 1.0_dp
                else
                    bin_fac(2) = bin_fac(2)*(deriv(2)-(m-1))/m
                end if
                ! toroidal derivatives
                do n = 0,deriv(3)
                    if (n.eq.0) then                                            ! first term in sum
                        bin_fac(3) = 1.0_dp
```

```
                    else
                        bin_fac(3) = bin_fac(3)*(deriv(3)-(n-1))/n
                    end if

                    ! current term in the tripple summation
                    arr_3 = arr_3 + &
                        &bin_fac(1)*bin_fac(2)*bin_fac(3) &
                        &* arr_1(:,:,:,r,m,n)&
                        &* arr_2(:,:,:,deriv(1)-r,deriv(2)-m,deriv(3)-n)
                end do
            end do
        end do
    end function add_arr_mult_3_3
    !> \private version with \c arr_1 in 3  coords and \c arr_2 only in the flux
    !! coord.
    integer function add_arr_mult_3_1(arr_1,arr_2,arr_3,deriv) result(ierr)
        character(*), parameter :: rout_name = 'add_arr_mult_3_1'

        ! input / output
        real(dp), intent(in) :: arr_1(1:,1:,1:,0:,0:,0:)                        !< \c arr_1
        real(dp), intent(in) :: arr_2(1:,0:)                                    !< \c arr_2
        real(dp), intent(out) :: arr_3(1:,1:,1:)                                !< \c arr_3
        integer, intent(in) :: deriv(3)                                         !< derivatives

        ! local variables
        real(dp) :: bin_fac                                                     ! binomial factor for norm. sum
        integer :: r                                                            ! current degree of norm. derivative
        integer :: kd                                                           ! normal counter
        character(len=max_str_ln) :: err_msg                                    ! error message

        ! initialize ierr
        ierr = 0

        ! tests
        if (size(arr_1,3).ne.size(arr_2,1)) then
            err_msg = 'Arrays 1 and 2 need to have the same size in the flux &
                &coord.'
            ierr = 1
            chckerr(err_msg)
        end if
        if (size(arr_1,1).ne.size(arr_3,1) .or. size(arr_1,2).ne.size(arr_3,2) &
            &.or. size(arr_1,3).ne.size(arr_3,3)) then
            err_msg = 'Array 1 needs to have the same size as the resulting &
                &array 3'
            ierr = 1
            chckerr(err_msg)
        end if

        ! distribute   normal  derivatives   using  binomial   theorem,  angular
        ! derivatives only operate on the first array
        ! normal derivatives
        do r = 0,deriv(1)
            if (r.eq.0) then                                                    ! first term in sum
                bin_fac = 1.0_dp
            else
                bin_fac = bin_fac*(deriv(1)-(r-1))/r
            end if

            ! current term in the tripple summation
            do kd = 1, size(arr_1,3)
                arr_3(:,:,kd) = arr_3(:,:,kd) + bin_fac * &
                    &arr_1(:,:,kd,r,deriv(2),deriv(3))* arr_2(kd,deriv(1)-r)
            end do
        end do
    end function add_arr_mult_3_1
    !> \private Version with \c arr_1 and \c arr_2 only in the flux coord.
    integer function add_arr_mult_1_1(arr_1,arr_2,arr_3,deriv) result(ierr)
        character(*), parameter :: rout_name = 'add_arr_mult_1_1'

        ! input / output
        real(dp), intent(in) :: arr_1(1:,0:)                                    !< \c arr_1
        real(dp), intent(in) :: arr_2(1:,0:)                                    !< \c arr_2
        real(dp), intent(out) :: arr_3(1:)                                      !< \c arr_3
        integer, intent(in) :: deriv(3)                                         !< derivatives

        ! local variables
        real(dp) :: bin_fac                                                     ! binomial factor for norm. sum
        integer :: r                                                            ! current degree of norm. derivative
        character(len=max_str_ln) :: err_msg                                    ! error message

        ! initialize ierr
        ierr = 0

        ! tests
        if (size(arr_1,1).ne.size(arr_2,1)) then
            err_msg = 'Arrays 1 and 2 need to have the same size in the &
                &flux coord.'
```

```
            ierr = 1
            chckerr(err_msg)
        end if
        if (size(arr_1,1).ne.size(arr_3,1)) then
            err_msg = 'Array 1 needs to have the same size as the resulting &
                &array 3'
            ierr = 1
            chckerr(err_msg)
        end if

        if (deriv(2).gt.0 .or. deriv(3).gt.0) then                          ! no addition to arr_3
            return
        else
            ! distribute  normal  derivatives  using binomial  theorem,  angular
            ! derivatives only operate on the first array
            ! normal derivatives
            do r = 0,deriv(1)
                if (r.eq.0) then                                                ! first term in sum
                    bin_fac = 1.0_dp
                else
                    bin_fac = bin_fac*(deriv(1)-(r-1))/r
                end if

                ! current term in the tripple summation
                arr_3 = arr_3 + bin_fac * arr_1(:,r)* arr_2(:,deriv(1)-r)
            end do
        end if
    end function add_arr_mult_1_1

    !> private array version
    integer recursive function calc_det_3d(deta,a,n) result (ierr)
        character(*), parameter :: rout_name = 'calc_det_3D'

        ! input / output
        real(dp), intent(inout) :: deta(:,:,:)                              !< output
        real(dp), intent(in) :: a(:,:,:,:)                                  !< input
        integer, intent(in) :: n                                            !< size of matrix

        ! local variables
        logical :: sym                                                      ! .true. if matrix symmetric
        integer :: id, jd, kd                                               ! counter
        integer :: nn                                                       ! nr. of elements in matrix
        real(dp) :: sgn                                                     ! holds either plus or minus one
        integer, allocatable :: slct(:)                                     ! 0 in between 1's selects which column to delete
        integer, allocatable :: idx(:)                                      ! counts from 1 to size(A)
        character(len=max_str_ln) :: err_msg                                ! error message
        real(dp), allocatable :: work(:,:,:)                                ! work array
        integer, allocatable :: c_sub(:)                                    ! indices of submatrix in storage convention of eq_vars.eq_2_type
        integer, allocatable :: k_sub(:)                                    ! first indices of submatrix in 2D

        ! initialize ierr
        ierr = 0

        ! tests
        if (size(deta,1).ne.size(a,1) .or. size(deta,2).ne.size(a,2) .or. &
            &size(deta,3).ne.size(a,3)) then
            err_msg = 'The output and input matrix have to have the same sizes'
            ierr = 1
            chckerr(err_msg)
        end if

        ! set total number of elements in matrix
        nn = size(a,4)

        ! set sym
        ierr = is_sym(n,nn,sym)
        chckerr('')

        ! intialize other quantities
        sgn = 1.0_dp
        allocate (idx(n))
        idx = [(id,id=1,n)]
        allocate (slct(n))
        slct = 1
        allocate (work(size(a,1),size(a,2),size(a,3)))

        deta = 0.0_dp

        if (n.eq.1) then                                                    ! shouldn't be used
            deta = a(:,:,:,c([1,1],sym,n))
        else if (n.eq.2) then
            deta = a(:,:,:,c([1,1],sym,n))*a(:,:,:,c([2,2],sym,n))-&
                &a(:,:,:,c([1,2],sym,n))*a(:,:,:,c([2,1],sym,n))
        else
            ! allocate coordinates of (n-1)x(n-1) submatrix
```

```
            allocate(c_sub((n-1)**2))
            allocate(k_sub(n**2)); k_sub = 0
            ! iterate over indices in second dimension
            do id = 1, n
                slct(id) = 0
                ! set up coordinates of submatrix (2:n,pack(idx,slct.gt.0))
                ! (in general not symmetric, even if parent matrix is)
                k_sub = pack(idx,slct.gt.0)
                do kd = 1,n-1
                    do jd = 1,n-1
                        c_sub((kd-1)*(n-1)+jd) = c([jd+1,k_sub(kd)],sym,n)
                    end do
                end do
                ierr = calc_det_3d(work,a(:,:,:,c_sub),n-1)
                chckerr('')
                deta = deta + a(:,:,:,c([1,id],sym,n))*sgn*work
                sgn = -sgn
                slct(id) = 1
            end do
            ! deallocate c_sub and k_sub
            deallocate(c_sub,k_sub)
        end if
    end function calc_det_3d
    !> private constant version
    integer function calc_det_0d(det_0D,A) result(ierr)
        character(*), parameter :: rout_name = 'calc_det_0D'

        ! input / output
        real(dp), intent(inout) :: det_0d                                   !< output
        real(dp), intent(in) :: a(:,:)                                      !< input

        ! local variables
        integer :: n
        real(dp) :: sgn
        integer :: id, info
        integer, allocatable :: ipiv(:)
        character(len=max_str_ln) :: err_msg                                ! error message
        real(dp), allocatable :: a_loc(:,:)                                 ! copy of A

        ! initialize ierr
        ierr = 0

        ! tests
        if (size(a,1).ne.size(a,2)) then
            err_msg = 'The matrix A has to be square'
            ierr = 1
            chckerr(err_msg)
        end if

        ! initialize
        n = size(a,1)
        allocate(ipiv(n))
        ipiv = 0

        ! set up local A
        allocate(a_loc(n,n))
        a_loc = a

        call dgetrf(n,n,a_loc,n,ipiv,info)

        det_0d = 1.0_dp
        do id = 1, n
            det_0d = det_0d*a_loc(id,id)
        end do

        sgn = 1.0_dp
        do id = 1, n
            if(ipiv(id).ne.id) then
                sgn = -sgn
            end if
        end do
        det_0d = sgn*det_0d

        ! deallocate local variables
        deallocate(a_loc)
    end function calc_det_0d

    !> \private array version
    integer function calc_inv_3d(inv_3D,A,n) result(ierr)
        use num_vars, only: tol_zero

        character(*), parameter :: rout_name = 'calc_inv_3D'

        ! input / output
        real(dp), intent(inout) :: inv_3d(:,:,:,:)                          !< output
        real(dp), intent(in) :: a(:,:,:,:)                                  !< input
        integer, intent(in) :: n                                            !< size of matrix
```

```fortran

        ! local variables
        logical :: sym                                                  ! .true. if matrix symmetric
        real(dp), allocatable :: deta(:,:,:)                            ! determinant of a submatrix of A
        integer :: id, jd, kd, ld                                       ! counters
        integer :: nn                                                   ! nr. of elements in matrix
        integer, allocatable :: slct(:,:)                               ! 0 in between 1's selects which column to delete
        integer, allocatable :: idx(:)                                  ! counts from 1 to size(A)
        character(len=max_str_ln) :: err_msg                            ! error message
        integer, allocatable :: c_sub(:)                                ! indices of submatrix in storage convention of eq_vars.eq_2_type
        integer, allocatable :: l_sub(:), k_sub(:)                      ! first and second indices of submatrix in 2D
        integer :: kd_min                                               ! min. of kd

        ! initialize ierr
        ierr = 0

        ! tests
        if (size(inv_3d,1).ne.size(a,1) .or. size(inv_3d,2).ne.size(a,2) .or. &
            &size(inv_3d,3).ne.size(a,3) .or. size(inv_3d,4).ne.size(a,4)) then
            err_msg = 'The output and input matrix have to have the same sizes'
            ierr = 1
            chckerr(err_msg)
        end if

        ! set total number of elements in matrix
        nn = size(a,4)

        ! set sym
        ierr = is_sym(n,nn,sym)
        chckerr('')

        ! initialize other quantitites
        allocate (idx(n))
        idx = [(id,id=1,n)]
        allocate (slct(n,2))
        slct = 1
        allocate(deta(size(a,1),size(a,2),size(a,3)))
        deta = 0.0_dp

        ! allocate coordinates of (n-1)x(n-1) submatrix
        allocate(c_sub((n-1)**2))
        allocate(l_sub(n**2)); l_sub = 0
        allocate(k_sub(n**2)); k_sub = 0

        ! set up kd_min
        kd_min = 1

        ! calculate cofactor matrix, replacing original elements
        ! use is made of detA
        do ld = 1,n
            ! adjust kd_min if symmetric
            if (sym) kd_min = ld
            do kd = kd_min,n
                ! set up coordinates of submatrix (strike out row i, column j)
                slct(ld,1) = 0                                          ! take out column ld
                slct(kd,2) = 0                                          ! take out row kd
                l_sub = pack(idx,slct(:,1).gt.0)
                k_sub = pack(idx,slct(:,2).gt.0)
                do jd = 1,n-1
                    do id = 1,n-1
                        c_sub((jd-1)*(n-1)+id) = c([l_sub(id),k_sub(jd)],sym,n) ! taking the transpose!
                    end do
                end do
                ierr = calc_det_3d(deta,a(:,:,:,c_sub),n-1)
                chckerr('')
                inv_3d(:,:,:,c([kd,ld],sym,n)) = (-1.0_dp)**(kd+ld)*deta
                slct(ld,1) = 1
                slct(kd,2) = 1
            end do
        end do

        ! deallocate c_sub l_sub and k_sub
        deallocate(c_sub,l_sub,k_sub)

        ! calculate determinant in detA
        ierr = calc_det(deta,a,n)
        chckerr('')

        ! limit determinant from being zero
        deta = sign(max(abs(deta),tol_zero),deta)

        ! divide by determinant
        do kd = 1,nn
            inv_3d(:,:,:,kd) = inv_3d(:,:,:,kd) / deta
```

```
        end do
    end function calc_inv_3d
    !> private constant version
    integer function calc_inv_0d(inv_0D,A) result(ierr)
        character(*), parameter :: rout_name = 'calc_inv_0D'

        ! input / output
        real(dp), intent(inout) :: inv_0d(:,:)                                  !< output
        real(dp), intent(in) :: a(:,:)                                          !< input

        ! local variables
        integer :: n
        integer, allocatable :: ipiv(:)                                         ! pivot variable, used by Lapack
        real(dp), allocatable :: w(:)                                           ! work variable
        character(len=max_str_ln) :: err_msg                                    ! error message

        ! initialize ierr
        ierr = 0

        ! tests
        if (size(a,1).ne.size(a,1)) then
            err_msg = 'The input matrix A has to be square'
            ierr = 1
            chckerr(err_msg)
        end if
        if (size(inv_0d,1).ne.size(a,1) .or. size(inv_0d,2).ne.size(a,2)) then
            err_msg = 'The output and input matrix have to have the same sizes'
            ierr = 1
            chckerr(err_msg)
        end if

        ! initialize
        n = size(a,1)
        allocate(ipiv(n))
        ipiv = 0
        allocate(w(n))

        inv_0d = a

        call dgetrf(n,n,inv_0d,n,ipiv,ierr)                                     ! LU factorization
        err_msg = 'Lapack couldn''t find the LU factorization'
        chckerr(err_msg)

        call dgetri(n,inv_0d,n,ipiv,w,n,ierr)                                   ! inverse of LU
        chckerr('Lapack couldn''t compute the inverse')
    end function calc_inv_0d

    !> \private real array version
    integer function calc_mult_3d_real(A,B,AB,n,transp) result(ierr)
        character(*), parameter :: rout_name = 'calc_mult_3D_real'

        ! input / output
        real(dp), intent(in) :: a(:,:,:,:)                                      !< input A
        real(dp), intent(in) :: b(:,:,:,:)                                      !< input B
        real(dp), intent(inout) :: ab(:,:,:,:)                                  !< output A B
        integer, intent(in) :: n                                                !< size of matrix
        logical, intent(in), optional :: transp(2)                              !< whether A and/or B transposed

        ! local variables
        logical :: sym(3)                                                       ! .true. if matrices symmetric
        integer :: nn(3)                                                        ! nr. of elements in matrices
        character(len=max_str_ln) :: err_msg                                    ! error message
        integer :: id, jd, kd                                                   ! counters
        integer :: id_min                                                       ! min. of id
        integer :: c1, c2, c3                                                   ! coordinates
        integer :: ind(2,3)                                                     ! indices
        logical :: transp_loc(2)                                                ! local copy of transp

        ! initialize ierr
        ierr = 0

        ! tests
        if (size(a,1).ne.size(b,1) .or. size(a,2).ne.size(b,2) .or. &
            &size(a,3).ne.size(b,3) .or. size(a,1).ne.size(ab,1) .or. &
            &size(a,2).ne.size(ab,2) .or. size(a,3).ne.size(ab,3)) then
            err_msg = 'The output and input matrices have to defined on the &
                &same grid'
            ierr = 1
            chckerr(err_msg)
        end if

        ! set local transp
        transp_loc = .false.
        if (present(transp)) transp_loc = transp

        ! set total number of elements in matrices
        nn(1) = size(a,4)
```

```
        nn(2) = size(b,4)
        nn(3) = size(ab,4)

        ! set sym
        do id = 1,3
            ierr = is_sym(n,nn(id),sym(id))
            chckerr('')
        end do

        ! initialize AB
        ab = 0._dp

        ! set up id_min
        id_min = 1

        ! loop over all 2D rows and columns of AB
        do jd = 1,n
            ! adjust id_min if AB symmetric
            if (sym(3)) id_min = jd
            do id = id_min,n
                do kd = 1,n
                    ind(:,3) = [id,jd]
                    c3 = c(ind(:,3),sym(3),n)
                    if (transp_loc(1)) then
                        ind(:,1) = [kd,id]
                    else
                        ind(:,1) = [id,kd]
                    end if
                    c1 = c(ind(:,1),sym(1),n)
                    if (transp_loc(2)) then
                        ind(:,2) = [jd,kd]
                    else
                        ind(:,2) = [kd,jd]
                    end if
                    c2 = c(ind(:,2),sym(2),n)
                    ab(:,:,:,c3) = ab(:,:,:,c3) + &
                        &a(:,:,:,c1)*b(:,:,:,c2)
                end do
            end do
        end do
    end function calc_mult_3d_real
    !> \private complex array version
    integer function calc_mult_3d_complex(A,B,AB,n,transp) result(ierr)
        character(*), parameter :: rout_name = 'calc_mult_3D_complex'

        ! input / output
        complex(dp), intent(in) :: a(:,:,:,:)                                   !< input A
        complex(dp), intent(in) :: b(:,:,:,:)                                   !< input B
        complex(dp), intent(inout) :: ab(:,:,:,:)                               !< output A B
        integer, intent(in) :: n                                                !< size of matrix
        logical, intent(in), optional :: transp(2)                              !< whether A and/or B transposed

        ! local variables
        logical :: sym(3)                                                       ! .true. if matrices symmetric
        integer :: nn(3)                                                        ! nr. of elements in matrices
        character(len=max_str_ln) :: err_msg                                    ! error message
        integer :: id, jd, kd                                                   ! counters
        integer :: id_min                                                       ! min. of id
        integer :: c1, c2, c3                                                   ! coordinates
        logical :: transp_loc(2)                                                ! local copy of transp
        integer :: ind(2,3)                                                     ! indices
        integer :: d(3)                                                         ! dimension of matrices A and B

        ! initialize ierr
        ierr = 0

        ! tests
        if (size(a,1).ne.size(b,1) .or. size(a,2).ne.size(b,2) .or. &
            &size(a,3).ne.size(b,3) .or. size(a,1).ne.size(ab,1) .or. &
            &size(a,2).ne.size(ab,2) .or. size(a,3).ne.size(ab,3)) then
            err_msg = 'The output and input matrices have to defined on the &
                &same grid'
            ierr = 1
            chckerr(err_msg)
        end if

        ! set local transp
        transp_loc = .false.
        if (present(transp)) transp_loc = transp

        ! set d
        d = [size(a,1),size(a,2),size(a,3)]

        ! set total number of elements in matrices
        nn(1) = size(a,4)
        nn(2) = size(b,4)
        nn(3) = size(ab,4)
```

```fortran

        ! set sym
        do id = 1,3
            ierr = is_sym(n,nn(id),sym(id))
            chckerr('')
        end do

        ! initialize AB
        ab = 0._dp

        ! set up id_min
        id_min = 1

        ! loop over all 2D rows and columns of AB
        do jd = 1,n
            ! adjust id_min if AB symmetric
            if (sym(3)) id_min = jd
            do id = id_min,n
                do kd = 1,n
                    ind(:,3) = [id,jd]
                    c3 = c(ind(:,3),sym(3),n)
                    if (transp_loc(1)) then
                        ind(:,1) = [kd,id]
                    else
                        ind(:,1) = [id,kd]
                    end if
                    c1 = c(ind(:,1),sym(1),n)
                    if (transp_loc(2)) then
                        ind(:,2) = [jd,kd]
                    else
                        ind(:,2) = [kd,jd]
                    end if
                    c2 = c(ind(:,2),sym(2),n)
                    ab(:,:,:,c3) = ab(:,:,:,c3) + &
                        &con(a(:,:,:,c1),ind(:,1),sym(1),d)*&
                        &con(b(:,:,:,c2),ind(:,2),sym(2),d)
                end do
            end do
        end do
    end function calc_mult_3d_complex
    !> \private real constant version
    integer function calc_mult_0d_real(A,B,AB,n,transp) result(ierr)
        character(*), parameter :: rout_name = 'calc_mult_0D_real'

        ! input / output
        real(dp), intent(in) :: a(:)                                    !< input A
        real(dp), intent(in) :: b(:)                                    !< input B
        real(dp), intent(inout) :: ab(:)                                !< output A B
        integer, intent(in) :: n                                        !< size of matrix
        logical, intent(in), optional :: transp(2)                      !< whether A and/or B transposed

        ! local variables
        real(dp), allocatable :: loc_ab(:,:,:,:)                        ! local copy of C

        ! initialize ierr
        ierr = 0

        ! allocate local AB
        allocate(loc_ab(1,1,1,size(ab)))

        ! calculate using 2D version
        ierr = calc_mult_3d_real(reshape(a,[1,1,1,size(a)]),&
            &reshape(b,[1,1,1,size(b)]),loc_ab,n,transp)

        ! update AB with local value
        ab = loc_ab(1,1,1,:)

        ! deallocate local variables
        deallocate(loc_ab)
    end function calc_mult_0d_real

    !> \private version defined on 3-D grid
    integer function conv_mat_3d(A,B,n,transp) result(ierr)
        character(*), parameter :: rout_name = 'conv_mat_3D'

        ! input / output
        real(dp), intent(in) :: a(:,:,:,:)                              !< matrix to be converted
        real(dp), intent(inout) :: b(:,:,:,:)                           !< converted matrix
        integer, intent(in) :: n                                        !< size of matrix
        logical, intent(in), optional :: transp                         !< transpose

        ! local variables
        logical :: sym(2)                                               ! .true. if matrices symmetric
        logical :: transp_loc                                           ! local transp
        integer :: nn(2)                                                ! nr. of elements in matrices
        integer :: id, jd                                               ! counters
        integer :: id_min                                               ! minimum value of id
```

```
        integer :: ind(2,2)                                             ! indices
        real(dp), allocatable :: a_loc(:,:,:,:)                         ! local copy of A

        ! initialize ierr
        ierr = 0

        ! set total number of elements in matrix
        nn(1) = size(a,4)
        nn(2) = size(b,4)

        ! set local transp
        transp_loc = .false.
        if (present(transp)) transp_loc = transp

        ! set local A
        allocate(a_loc(size(a,1),size(a,2),size(a,3),size(a,4)))
        a_loc = a

        ! set sym
        do id = 1,2
            ierr = is_sym(n,nn(id),sym(id))
            chckerr('')
        end do

        ! copy A into B
        if (sym(1).and.sym(2) .or. &
            &.not.sym(1).and..not.sym(2).and..not.transp_loc) then      ! both symmetric or both unsymmetric and no transpose needed
            b = a                                                       ! symmetric matrix equal to transpose or no tranpose needed
        else                                                            ! at least one is not symmetric
            ! loop over row
            do jd = 1,n
                ! set id_min
                if (sym(2)) then
                    id_min = jd
                else
                    id_min = 1
                end if
                ! loop over column
                do id = id_min,n
                    ! set ind
                    if (transp_loc) then
                        ind(:,1) = [jd,id]
                    else
                        ind(:,1) = [id,jd]
                    end if
                    ind(:,2) = [id,jd]
                    ! copy elements of A into B
                    b(:,:,:,c(ind(:,2),sym(2),n)) = &
                        &a_loc(:,:,:,c(ind(:,1),sym(1),n))
                end do
            end do
        end if
    end function conv_mat_3d
    !> \private scalar version
    integer function conv_mat_0d(A,B,n,transp) result(ierr)
        character(*), parameter :: rout_name = 'conv_mat_0D'

        ! input / output
        real(dp), intent(in) :: a(:)                                    !< matrix to be converted
        real(dp), intent(inout) :: b(:)                                 !< converted matrix
        integer, intent(in) :: n                                        !< size of matrix
        logical, intent(in), optional :: transp                         !< transpose

        ! local variables
        real(dp), allocatable :: b_loc(:,:,:,:)                         ! local B

        ! initialize ierr
        ierr = 0

        ! set up local B
        allocate(b_loc(1,1,1,size(b)))

        ! call 3D version
        ierr = conv_mat_3d(reshape(a,[1,1,1,size(a)]),b_loc,n,transp)
        chckerr('')

        ! copy to B
        b = b_loc(1,1,1,:)
    end function conv_mat_0d
    !> \private complex version defined on 3D grid
    integer function conv_mat_3d_complex(A,B,n,transp) result(ierr)
        character(*), parameter :: rout_name = 'conv_mat_3D_complex'

        ! input / output
        complex(dp), intent(in) :: a(:,:,:,:)                           !< matrix to be converted
```

```
        complex(dp), intent(inout) :: b(:,:,:,:)                        !< converted matrix
        integer, intent(in) :: n                                        !< size of matrix
        logical, intent(in), optional :: transp                         !< transpose

        ! local variables
        logical :: sym(2)                                               ! .true. if matrices symmetric
        logical :: transp_loc                                           ! local transp
        integer :: nn(2)                                                ! nr. of elements in matrices
        integer :: id, jd                                               ! counters
        integer :: id_min                                               ! minimum value of id
        integer :: ind(2,2)                                             ! indices
        complex(dp), allocatable :: a_loc(:,:,:,:)                      ! local copy of A

        ! initialize ierr
        ierr = 0

        ! set total number of elements in matrix
        nn(1) = size(a,4)
        nn(2) = size(b,4)

        ! set local transp
        transp_loc = .false.
        if (present(transp)) transp_loc = transp

        ! set local A
        allocate(a_loc(size(a,1),size(a,2),size(a,3),size(a,4)))
        a_loc = a

        ! set sym
        do id = 1,2
            ierr = is_sym(n,nn(id),sym(id))
            chckerr('')
        end do

        ! copy A into B
        if (sym(1).and.sym(2) .or. &
            &.not.sym(1).and..not.sym(2).and..not.transp_loc) then       ! both symmetric or both unsymmetric and no transpose needed
            b = a                                                       ! symmetric matrix equal to transpose or no tranpose needed
        else                                                            ! at least one is not symmetric
            ! loop over row
            do jd = 1,n
                ! set id_min
                if (sym(2)) then
                    id_min = jd
                else
                    id_min = 1
                end if
                ! loop over column
                do id = id_min,n
                    ! set ind
                    if (transp_loc) then
                        ind(:,1) = [jd,id]
                    else
                        ind(:,1) = [id,jd]
                    end if
                    ind(:,2) = [id,jd]
                    ! copy elements of A into B
                    b(:,:,:,c(ind(:,2),sym(2),n)) = &
                        &a_loc(:,:,:,c(ind(:,1),sym(1),n))
                end do
            end do
        end if
    end function conv_mat_3d_complex
    !> \private complex scalar version
    integer function conv_mat_0d_complex(A,B,n,transp) result(ierr)
        character(*), parameter :: rout_name = 'conv_mat_0D_complex'

        ! input / output
        complex(dp), intent(in) :: a(:)                                 !< matrix to be converted
        complex(dp), intent(inout) :: b(:)                              !< converted matrix
        integer, intent(in) :: n                                        !< size of matrix
        logical, intent(in), optional :: transp                         !< transpose

        ! local variables
        complex(dp), allocatable :: b_loc(:,:,:,:)                      ! local B

        ! initialize ierr
        ierr = 0

        ! set up local B
        allocate(b_loc(1,1,1,size(b)))

        ! call 3D version
        ierr = conv_mat_3d_complex(reshape(a,[1,1,1,size(a)]),b_loc,n,transp)
        chckerr('')
```

```

        ! copy to B
        b = b_loc(1,1,1,:)
    end function conv_mat_0d_complex

    !> \private equidistant version
    integer function con2dis_eqd(pt_c,pt_d,lim_c,lim_d) result(ierr)
        !character(*), parameter :: rout_name = 'con2dis_eqd'

        ! input / output
        real(dp), intent(in) :: pt_c                                    !< point on continous grid
        real(dp), intent(inout) :: pt_d                                 !< point on discrete grid
        real(dp), intent(in) :: lim_c(2)                                !< <tt>[min_c,max_c]</tt>
        integer, intent(in) :: lim_d(2)                                 !< <tt>[min_d,max_d]</tt>

        ! initialize ierr
        ierr = 0

        ! Calculate, using the formula:
        !    pt_c - min_c     pt_d - min_d
        !   ------------- =  ------------- ,
        !   max_c - min_c    max_d - min_d
        pt_d = lim_d(1) + (lim_d(2)-lim_d(1)) * (pt_c-lim_c(1)) / &
            &(lim_c(2)-lim_c(1))
    end function con2dis_eqd
    !> \private regular version
    integer function con2dis_reg(pt_c,pt_d,var_c) result(ierr)
        character(*), parameter :: rout_name = 'con2dis_reg'

        ! input / output
        real(dp), intent(in) :: pt_c                                    !< point on continous grid
        real(dp), intent(inout) :: pt_d                                 !< point on discrete grid
        real(dp), intent(in) :: var_c(:)                                !< continous grid values

        ! local variables
        real(dp), allocatable :: var_c_loc(:)                           ! local copy of var
        real(dp), allocatable :: var_c_inv(:)                           ! inverted var_c
        integer :: size_c                                               ! size of var_c
        integer :: ind_lo, ind_hi                                       ! lower and upper index comprising pt_c
        integer :: id                                                   ! counter
        real(dp) :: pt_c_loc                                            ! local pt_c
        character(len=max_str_ln) :: err_msg                            ! error message
        real(dp), parameter :: tol = 1.e-8_dp                           ! tolerance for comparisons

        ! initialize ierr
        ierr = 0

        ! set up size_c
        size_c = size(var_c)

#if ldebug
        if (debug_con2dis_reg) &
            &call writo('finding the discrete index of '//trim(r2str(pt_c)))
#endif

        ! set up local var_c and pt_c_loc
        allocate(var_c_loc(size_c))
        if (var_c(1).lt.var_c(size_c)) then
            var_c_loc = var_c
            pt_c_loc = pt_c
        else
#if ldebug
            if (debug_con2dis_reg) &
                &call writo('setting var_c_loc = - var_c and pt_c_loc = -pt_c')
#endif
            var_c_loc = - var_c                                         ! local var should be rising
            pt_c_loc = - pt_c                                           ! invert pt_c as well
        end if

        ! set up inverse var_c
        allocate(var_c_inv(size_c))
        var_c_inv = var_c_loc(size_c:1:-1)

        ! find the lower and upper index comprising local pt_c
        ind_lo = 0
        ind_hi = size_c+1
        do id = 1,size_c
            if (pt_c_loc+tol*abs(pt_c_loc) .ge. var_c_loc(id)) then
                ind_lo = id
#if ldebug
                if (debug_con2dis_reg) call writo('for iteration '//&
                    &trim(i2str(id))//'/'//trim(i2str(size_c))//', '//&
                    &trim(r2strt(pt_c_loc))//' + '//trim(r2strt(tol))//&
                    &'*abs('//trim(r2strt(pt_c_loc))//') >= '//&
                    &trim(r2strt(var_c_loc(id)))//' => ['//&
                    &trim(i2str(ind_lo))//','//trim(i2str(ind_hi))//']')
#endif
```

```
            end if
            if (pt_c_loc-tol*abs(pt_c_loc) .le. var_c_inv(id)) then
                ind_hi = size_c+1-id
#if ldebug
                if (debug_con2dis_reg) call writo('for iteration '//&
                    &trim(i2str(id))//'/'//trim(i2str(size_c))//', '//&
                    &trim(r2strt(pt_c_loc))//' - '//trim(r2strt(tol))//&
                    &'*abs('//trim(r2strt(pt_c_loc))//') < '//&
                    &trim(r2strt(var_c_inv(id)))//' => ['//&
                    &trim(i2str(ind_lo))//','//trim(i2str(ind_hi))//']')
#endif
            end if
        end do
#if ldebug
        if (debug_con2dis_reg) then
            call writo('final ind_lo = '//trim(i2str(ind_lo))//', ind_hi = '//&
                &trim(i2str(ind_hi)))
            !!call print_ex_2D('var_c_loc, var_c_inv','',&
                !!&reshape([var_c_loc,var_c_inv],[size_c,2]))
        end if
#endif

        ! tests
        if (ind_lo.eq.0 .or. ind_hi.eq.size_c+1) then                          ! not within range
            pt_d = -1._dp
            ierr = 1
            err_msg = 'pt_c '//trim(r2str(pt_c))//' not within range '//&
                &trim(r2str(minval(var_c)))//'..'//trim(r2str(maxval(var_c)))
            chckerr(err_msg)
        end if

        ! set output
        if (ind_lo.lt.ind_hi) then                                             ! valid output that does not correspond to a point on grid
            pt_d = ind_lo + (pt_c_loc-var_c_loc(ind_lo))/&
                &(var_c_loc(ind_hi)-var_c_loc(ind_lo))
        else if (ind_lo.eq.ind_hi) then                                        ! valid output that does correspond to a point on grid
            pt_d = ind_lo
        else                                                                   ! invalid output
            pt_d = -1._dp
            ierr = 1
            err_msg = 'ind_lo cannot be higher than ind_hi'
            chckerr(err_msg)
        end if

        ! deallocate
        deallocate(var_c_loc,var_c_inv)
    end function con2dis_reg

    !> \private equidistant version
    integer function dis2con_eqd(pt_d,pt_c,lim_d,lim_c) result(ierr)
        !character(*), parameter :: rout_name = 'dis2con_eqd'

        ! input / output
        integer, intent(in) :: pt_d                                            !< point on discrete grid
        real(dp), intent(inout) :: pt_c                                        !< point on continous grid
        integer, intent(in) :: lim_d(2)                                        !< <tt>[min_d,max_d]</tt>
        real(dp), intent(in) :: lim_c(2)                                       !< <tt>[min_c,max_c]</tt>

        ! initialize ierr
        ierr = 0

        ! Calculate, using the formula:
        !    pt_c - min_c     pt_d - min_d
        !   ------------- =  ------------- ,
        !   max_c - min_c    max_d - min_d
        pt_c = lim_c(1) + (lim_c(2)-lim_c(1)) * (pt_d-lim_d(1)) / &
            &(lim_d(2)-lim_d(1))
    end function dis2con_eqd
    !> \private regular version
    integer function dis2con_reg(pt_d,pt_c,var_c) result(ierr)
        character(*), parameter :: rout_name = 'dis2con_reg'

        ! input / output
        integer, intent(in) :: pt_d                                            !< point on discrete grid
        real(dp), intent(inout) :: pt_c                                        !< point on continous grid
        real(dp), intent(in) :: var_c(:)                                       !< continous grid values

        ! local variables
        character(len=max_str_ln) :: err_msg                                   ! error message

        ! initialize ierr
        ierr = 0

        ! Check whether the discrete value lies inside the range
        if (pt_d.lt.1 .or. pt_d.gt.size(var_c)) then
```

```
            pt_c = -1._dp
            ierr = 1
            err_msg = 'pt_c not within range'
            chckerr(err_msg)
        end if

        ! Return the continuous variable
        pt_c = var_c(pt_d)
    end function dis2con_reg

    !> \private array version
    integer function round_with_tol_arr(vals,lim_lo,lim_hi,tol) result(ierr)
        character(*), parameter :: rout_name = 'round_with_tol_arr'

        ! input / output
        real(dp), intent(inout) :: vals(:)                                  !< values to be rounded
        real(dp), intent(in) :: lim_lo                                      !< lower limit on val
        real(dp), intent(in) :: lim_hi                                      !< upper limit on val
        real(dp), intent(in), optional :: tol                               !< tolerance

        ! local variables
        real(dp) :: loc_tol                                                 ! local value of tol
        character(len=max_str_ln) :: err_msg                                ! error message

        ! initialize ierr
        ierr = 0

        ! set loc_tol
        if (present(tol)) then
            loc_tol = tol
        else
            loc_tol = 1e-5_dp
        end if

        ! set possible error message
        err_msg = 'value has to lie within 0..1'

        if (minval(vals-lim_lo+loc_tol).lt.0) then
            ierr = 1
            chckerr(err_msg)
        else if (maxval(vals-lim_hi-loc_tol).gt.0) then
            ierr = 1
            chckerr(err_msg)
        else
            where (vals.lt.lim_lo) vals = lim_lo
            where (vals.gt.lim_hi) vals = lim_hi
        end if
    end function round_with_tol_arr
    !> \private individual version
    integer function round_with_tol_ind(val,lim_lo,lim_hi,tol) result(ierr)
        character(*), parameter :: rout_name = 'round_with_tol_ind'

        ! input / output
        real(dp), intent(inout) :: val                                      !< value to be rounded
        real(dp), intent(in) :: lim_lo                                      !< lower limit on val
        real(dp), intent(in) :: lim_hi                                      !< upper limit on val
        real(dp), intent(in), optional :: tol                               !< tolerance

        ! local variables
        real(dp) :: vals(1)                                                 ! array copy of val

        ! initialize ierr
        ierr = 0

        vals(1) = val

        ierr = round_with_tol_arr(vals,lim_lo,lim_hi,tol)
        chckerr('')

        val = vals(1)
    end function round_with_tol_ind

    !> \private 3-D version
    function con_3d(A,c,sym,d) result(B)
        complex(dp), intent(in) :: a(:,:,:)                                 !< input A
        integer, intent(in) :: c(2)                                         !< indices in square matrix
        logical, intent(in) :: sym                                          !< if A is symmetric
        integer, intent(in) :: d(3)                                         !< dimensions of matrix A
        complex(dp) :: b(d(1),d(2),d(3))                                    !< output B

        ! determine whether to take complex conjugate or not
        if (c(2).gt.c(1) .and. sym) then                                    ! upper triangular matrix
            b = conjg(a)
        else
            b = a
        end if
    end function con_3d
```

```
    !> \private 2-D version
    function con_2d(A,c,sym,d) result(B)
        complex(dp), intent(in) :: a(:,:)                                   !< input A
        integer, intent(in) :: c(2)                                         !< indices in square matrix
        logical, intent(in) :: sym                                          !< if A is symmetric
        integer, intent(in) :: d(2)                                         !< dimensions of matrix A
        complex(dp) :: b(d(1),d(2))                                         !< output B

        ! determine whether to take complex conjugate or not
        if (c(2).gt.c(1) .and. sym) then                                    ! upper triangular matrix
            b = conjg(a)
        else
            b = a
        end if
    end function con_2d
    !> \private 1-D version
    function con_1d(A,c,sym,d) result(B)
        complex(dp), intent(in) :: a(:)                                     !< input A
        integer, intent(in) :: c(2)                                         !< indices in square matrix
        logical, intent(in) :: sym                                          !< if A is symmetric
        integer, intent(in) :: d(1)                                         !< dimensions of matrix A
        complex(dp) :: b(d(1))                                              !< output B

        ! determine whether to take complex conjugate or not
        if (c(2).gt.c(1) .and. sym) then                                    ! upper triangular matrix
            b = conjg(a)
        else
            b = a
        end if
    end function con_1d
    !> \private 0-D version
    function con_0d(A,c,sym) result(B)
        complex(dp), intent(in) :: a                                        !< input A
        integer, intent(in) :: c(2)                                         !< indices in square matrix
        logical, intent(in) :: sym                                          !< if A is symmetric
        complex(dp) :: b                                                    !< output B

        ! determine whether to take complex conjugate or not
        if (c(2).gt.c(1) .and. sym) then                                    ! upper triangular matrix
            b = conjg(a)
        else
            b = a
        end if
    end function con_0d

    !> \private real version
    subroutine bubble_sort_real(a,piv)
        ! input / output
        real(dp), intent(inout) :: a(:)                                     !< vector to sort
        integer, intent(inout), optional :: piv(:)                          !< pivots

        ! local variables
        real(dp) :: temp                                                    ! temporary value
        integer :: id, jd                                                   ! counter
        logical :: swapped                                                  ! value swapped with next
        logical :: save_piv                                                 ! save pivots

        save_piv = .false.
        if (present(piv)) then
            if (size(piv).eq.size(a)) then
                save_piv = .true.
                piv = [(jd,jd=1,size(a))]
            end if
        end if

        do jd = size(a)-1,1,-1
            swapped = .false.
            do id = 1, jd
                if (a(id) .gt. a(id+1)) then
                    temp = a(id)
                    a(id) = a(id+1)
                    a(id+1) = temp
                    piv(id:id+1) = piv(id+1:id:-1)
                    swapped = .true.
                end if
            end do
            if (.not. swapped) exit
        end do
    end subroutine bubble_sort_real
    !> \private integer version
    subroutine bubble_sort_int(a,piv)
        ! input / output
        integer, intent(inout) :: a(:)                                      !< vector to sort
        integer, intent(inout), optional :: piv(:)                          !< pivots

        ! local variables
        integer :: temp                                                     ! temporary value
```

```
        integer :: id, jd                                                   ! counter
        logical :: swapped                                                  ! value swapped with next
        logical :: save_piv                                                 ! save pivots

        save_piv = .false.
        if (present(piv)) then
            if (size(piv).eq.size(a)) then
                save_piv = .true.
                piv = [(jd,jd=1,size(a))]
            end if
        end if

        do jd = size(a)-1,1,-1
            swapped = .false.
            do id = 1, jd
                if (a(id) .gt. a(id+1)) then
                    temp = a(id)
                    a(id) = a(id+1)
                    a(id+1) = temp
                    if (present(piv)) piv(id:id+1) = piv(id+1:id:-1)
                    swapped = .true.
                end if
            end do
            if (.not. swapped) exit
        end do
    end subroutine bubble_sort_int

    !> \private version with \c x and \c y separate
    integer function order_per_fun_1(x,y,x_out,y_out,overlap,tol) result(ierr)
        use num_vars, only: tol_zero

        character(*), parameter :: rout_name = 'order_per_fun_1'

        ! input / output
        real(dp), intent(in) :: x(:)                                        !< abscissa
        real(dp), intent(in) :: y(:)                                        !< ordinate
        real(dp), intent(inout), allocatable :: x_out(:)                    !< ordered x
        real(dp), intent(inout), allocatable :: y_out(:)                    !< ordered y
        integer, intent(in) :: overlap                                      !< overlap to include
        real(dp), intent(in), optional :: tol                               !< tolerance for error

        ! local variables
        real(dp) :: tol_loc                                                 ! local tolerance
        real(dp), allocatable :: x_fund(:)                                  ! x on fundamental interval 0..2pi
        integer :: ml_x                                                     ! location of maximum of x
        integer :: lim(2)                                                   ! limits in x
        integer :: lim_loc(2)                                               ! limits in loc_x
        character(len=max_str_ln) :: err_msg                                ! error message

        ! initialize ierr
        ierr = 0

        ! set local tol
        tol_loc = tol_zero
        if (present(tol)) tol_loc = tol

        ! tests
        if (abs(cos(x(size(x)))-cos(x(1))).lt.tol_loc .and. &
            &abs(sin(x(size(x)))-sin(x(1))).lt.tol_loc) then
            ierr = 1
            err_msg = 'First and last point cannot be identical. Remove one.'
            chckerr(err_msg)
        end if

        ! initialize
        allocate(x_fund(size(x)))
        x_fund = mod(x,2*pi)
        where(x_fund.lt.0._dp) x_fund = x_fund+2*pi
        ml_x = maxloc(x_fund,1)

        allocate(x_out(size(x)+2*overlap))
        allocate(y_out(size(x)+2*overlap))

        ! left overlap
        lim = [max(ml_x-overlap+1,1),ml_x]                                  ! all possible points to the left of ml_x, u
        lim_loc = [overlap-lim(2)+lim(1),overlap]
        x_out(lim_loc(1):lim_loc(2)) = x_fund(lim(1):lim(2))-2*pi
        y_out(lim_loc(1):lim_loc(2)) = y(lim(1):lim(2))
        lim_loc = [1,lim_loc(1)-1]
        lim = [size(x)-(lim_loc(2)-lim_loc(1)),size(x)]
        x_out(lim_loc(1):lim_loc(2)) = x_fund(lim(1):lim(2))-2*pi
        y_out(lim_loc(1):lim_loc(2)) = y(lim(1):lim(2))

        ! bulk
        lim = [ml_x+1,size(x)]                                              ! points after max, until last
        lim_loc = [overlap+1,overlap+lim(2)-lim(1)+1]
        x_out(lim_loc(1):lim_loc(2)) = x_fund(lim(1):lim(2))
```

```
        y_out(lim_loc(1):lim_loc(2)) = y(lim(1):lim(2))
        lim = [1,ml_x]                                                  ! points from start, until max
        lim_loc = lim_loc(2) + [1,lim(2)-lim(1)+1]
        x_out(lim_loc(1):lim_loc(2)) = x_fund(lim(1):lim(2))
        y_out(lim_loc(1):lim_loc(2)) = y(lim(1):lim(2))

        ! right overlap
        lim = [ml_x+1,min(ml_x+overlap,size(x))]                        ! all possible poinst to the right of ml_X
        lim_loc = lim_loc(2) + [1,lim(2)-lim(1)+1]
        x_out(lim_loc(1):lim_loc(2)) = x_fund(lim(1):lim(2))+2*pi
        y_out(lim_loc(1):lim_loc(2)) = y(lim(1):lim(2))
        lim_loc = [lim_loc(2)+1,size(x)+2*overlap]                      ! remaining point loop around
        lim = [1,lim_loc(2)-lim_loc(1)+1]
        x_out(lim_loc(1):lim_loc(2)) = x_fund(lim(1):lim(2))+2*pi
        y_out(lim_loc(1):lim_loc(2)) = y(lim(1):lim(2))
    end function order_per_fun_1
    !> \private version with \c x and \c y together
    integer function order_per_fun_2(xy,xy_out,overlap,tol) result(ierr)
        character(*), parameter :: rout_name = 'order_per_fun_2'

        ! input / output
        real(dp), intent(in) :: xy(:,:)                                 !< abscissa and ordinate
        real(dp), intent(inout), allocatable :: xy_out(:,:)             !< ordered xy
        integer, intent(in) :: overlap                                  !< overlap to include
        real(dp), intent(in), optional :: tol                           !< tolerance for error

        ! local variables
        real(dp), allocatable :: x_out(:)                               ! local x_out
        real(dp), allocatable :: y_out(:)                               ! local y_out

        ! initialize ierr
        ierr = 0

        ierr = order_per_fun_1(xy(:,1),xy(:,2),x_out,y_out,overlap,tol)
        chckerr('')

        allocate(xy_out(size(x_out),2))
        xy_out(:,1) = x_out
        xy_out(:,2) = y_out
    end function order_per_fun_2

    !> \private real version
    integer function spline_real(x,y,xnew,ynew,ord,deriv,bcs,bcs_val,extrap) &
        &result(ierr)

        use ezspline_obj
        use ezspline

        character(*), parameter :: rout_name = 'spline_real'

        ! input / output
        real(dp), intent(in), target :: x(:)                            !< coordinates
        real(dp), intent(in) :: y(:)                                    !< function value
        real(dp), intent(in), target :: xnew(:)                         !< new coordinates
        real(dp), intent(out) :: ynew(:)                                !< new function values
        integer, intent(in), optional :: ord                            !< order [def 3]
        integer, intent(in), optional :: deriv                          !< derivative [def 0]
        integer, intent(in), optional :: bcs(2)                         !< boundary conditions [def 0]
        real(dp), intent(in), optional :: bcs_val(2)                    !< boundary conditions [no def]
        logical, intent(in), optional :: extrap                         !< whether extrapolation is allowed [def
.false.]

        ! local variables
        type(ezspline1_r8) :: f_spl                                     ! spline object
        integer :: ord_loc                                              ! local order
        integer :: deriv_loc                                            ! local deriv
        integer :: bcs_loc(2)                                           ! local bcs
        integer :: kd                                                   ! counter
        integer :: n                                                    ! size of x, y
        integer :: nnew                                                 ! size of output x, y
        integer :: nnew_interp                                          ! size of interpolated x, y, rest is
extrapolated
        integer :: il(2)                                                ! limits of interp., what falls outside is
extrapolated
        real(dp), pointer :: x_loc(:)                                   ! x or -x
        real(dp), pointer :: xnew_loc(:)                                ! xnew or -xnew
        real(dp) :: bcs_val_loc(2)                                      ! local bcs_val
        real(dp) :: lim_vals(0:3)                                       ! limit values at boundaries for
extrapolation
        real(dp), allocatable :: xnew_ez(:)                             ! xnew in EZ spline doubles
        real(dp), allocatable :: ynew_ez(:)                             ! ynew in EZ spline doubles
        character(len=max_str_ln) :: err_msg                            ! error message
        logical :: flip_x                                               ! whether x axis is flipped
        logical :: extrap_loc                                           ! local extrap

        ! initialize ierr
        ierr = 0
```

```

        ! set local variables
        ord_loc = 3
        if (present(ord)) ord_loc = ord
        deriv_loc = 0
        if (present(deriv)) deriv_loc = deriv
        bcs_loc = [0,0]
        if (present(bcs)) bcs_loc = bcs
        if (present(bcs_val)) bcs_val_loc = bcs_val
        extrap_loc = .false.
        if (present(extrap)) extrap_loc = extrap

        ! set other variables
        n = size(x)
        nnew = size(xnew)

        ! set local x and xnew, possibly flipped if negative
        flip_x = (x(2) .lt. x(1))
        if (flip_x) then
            allocate(x_loc(n))
            allocate(xnew_loc(nnew))
            x_loc = -x
            xnew_loc = -xnew
        else
            x_loc => x
            xnew_loc => xnew
        end if

#if ldebug
        ! check monotony
        do kd = 2,n
            if (x_loc(kd).le.x_loc(kd-1)) then
                ierr = 1
                err_msg = '|x| is not monotonously increasing'
                chckerr(err_msg)
            end if
        end do

        ! check array sizes
        if (size(y).ne.n) then
            ierr = 1
            err_msg = 'x and y need to have the same size'
            chckerr(err_msg)
        end if
        if (size(ynew).ne.nnew) then
            ierr = 1
            err_msg = 'xnew and ynew need to have the same size'
            chckerr(err_msg)
        end if

        ! check order
        if (ord_loc.lt.1 .or. ord_loc.gt.3) then
            ierr = 1
            err_msg = 'Only order 1 (linear), 2 (akima hermite) or 3 (cubic) &
                &possible'
            chckerr(err_msg)
        end if

        ! check derviative
        select case (deriv_loc)
            case (:-1)
                ierr = 1
                err_msg = 'only nonnegative degrees of derivative are possible'
                chckerr(err_msg)
            case (0:1)
                ! do nothing
            case (2:3)
                if (ord_loc.eq.1 .or. ord_loc.eq.2) then
                    ierr = 1
                    err_msg = 'Derivative of degree '//trim(i2str(deriv_loc))//&
                        &' not possible for order '//trim(i2str(ord_loc))
                    chckerr(err_msg)
                end if
            case (4:)
                ierr = 1
                err_msg = 'maximum degree of derivative is 2 for order 4'
                chckerr(err_msg)
        end select

        ! check boundary conditions
        select case (ord_loc)
            case (1)
                if (present(bcs) .or. present(bcs_val)) then
                    call writo('for order 1, no boundary conditions can be &
                        &prescribed',alert=.true.)
                end if
            case (2:3)
```

```
                ! check possibility
                do kd = 1,2
                    if (bcs_loc(kd).lt.-1 .or. bcs_loc(kd).gt.ord_loc-1) then
                        ierr = 1
                        err_msg = 'For order '//trim(i2str(ord_loc))//&
                            &' bcs has to be -1..'//trim(i2str(ord_loc-1))
                        chckerr(err_msg)
                    end if
                end do

                ! check whether derivatives are provided
                if ((any(bcs_loc.eq.1) .or. any(bcs_loc.eq.2)) &
                    &.and. .not.present(bcs_val)) then
                    ierr = 1
                    err_msg = 'When prescribing first or second derviatives, &
                        &need to provide bcs_val'
                    chckerr(err_msg)
                end if
        end select
#endif

        ! set up interpolation limits
        do kd = 1,nnew
            if (xnew_loc(kd).ge.x_loc(1)) exit
        end do
        il(1) = kd
        do kd = nnew,1,-1
            if (xnew_loc(kd).le.x_loc(n)) exit
        end do
        il(2) = kd
        nnew_interp = il(2)-il(1)+1

        ! check for extrapolation
        if ((il(1).ne.1 .or. il(2).ne.nnew) .and. &
            &.not.extrap_loc) then
            ierr = 1
            call writo('xnew = ['//trim(r2str(minval(xnew)))//'..'//&
                &trim(r2str(maxval(xnew)))//']')
            call writo('but x = ['//trim(r2str(minval(x)))//'..'//&
                &trim(r2str(maxval(x)))//']')
            err_msg = 'Extrapolation needed, but not allowed'
            chckerr(err_msg)
        end if

        ! initialize
        if (ord_loc.eq.1) then
            call ezlinear_init(f_spl,n,ierr)
            call ezspline_error(ierr)
            chckerr('')
        else
            call ezspline_init(f_spl,n,bcs_loc,ierr)
            call ezspline_error(ierr)
            chckerr('')
            if (ord_loc.eq.2) f_spl%isHermite = 1
        end if

        ! set grid
        f_spl%x1 = x_loc

        ! set boundary condition
        if (present(bcs_val)) then
            f_spl%bcval1min = bcs_val_loc(1)
            f_spl%bcval1max = bcs_val_loc(2)
            if (flip_x) then                                                ! if x-axis flipped, also first derivative flipped
                if (bcs_loc(1) .eq. 1) f_spl%bcval1min = -f_spl%bcval1min
                if (bcs_loc(2) .eq. 1) f_spl%bcval1max = -f_spl%bcval1max
            end if
        end if

        ! set up
        call ezspline_setup(f_spl,y,ierr,exact_dim=.true.)                  ! match exact dimensions
        call ezspline_error(ierr)
        chckerr('')
        ! interpolated part
        if (il(1).le.il(2)) then
            allocate(xnew_ez(nnew_interp))
            allocate(ynew_ez(nnew_interp))
            xnew_ez = xnew_loc(il(1):il(2))

            if (deriv_loc.eq.0) then
                ! interpolate
                call ezspline_interp(f_spl,nnew_interp,xnew_ez,ynew_ez,ierr)
                call ezspline_error(ierr)
                chckerr('')
            else
                call ezspline_derivative(f_spl,deriv_loc,nnew_interp,xnew_ez,&
```

```
                    &ynew_ez,ierr)
                call ezspline_error(ierr)
                chckerr('')
            end if

            ynew(il(1):il(2)) = ynew_ez
            deallocate(xnew_ez,ynew_ez)
        end if

        ! extrapolated part
        if (extrap_loc) then
            ! extrapolate left
            if (il(1).gt.1) then
                ! set up limit values at first point
                ierr = setup_lim_vals(x_loc(1),lim_vals)
                chckerr('')

                ! calculate extrapolation
                call calc_extrap(xnew_loc(1:il(1)-1),x_loc(1),lim_vals,&
                    &ynew(1:il(1)-1))
            end if

            ! extrapolate right
            if (il(2).lt.nnew) then
                ! set up limit values at last point
                ierr = setup_lim_vals(x_loc(n),lim_vals)
                chckerr('')

                ! calculate extrapolation
                call calc_extrap(xnew_loc(il(2)+1:nnew),x_loc(n),lim_vals,&
                    &ynew(il(2)+1:nnew))
            end if
        end if

        ! free
        call ezspline_free(f_spl,ierr)
        chckerr('')
        call ezspline_error(ierr)
        chckerr('')
        if (flip_x) then
            deallocate(x_loc)
            deallocate(xnew_loc)
        end if
        nullify(x_loc)
        nullify(xnew_loc)
    contains
        !> /private set up limit values
        !!
        !! Makes use of f_spl
        integer function setup_lim_vals(xb,lim_vals) result(ierr)
            character(*), parameter :: rout_name = 'setup_lim_vals'

            ! input / output
            real(dp), intent(in) :: xb                                  ! x of boundary
            real(dp), intent(out) :: lim_vals(0:3)                      ! limit values

            ! local variables
            integer :: kd                                               ! counter
            integer :: max_deriv                                        ! maximum degree of derivative

            ! initialize ierr
            ierr = 0

            ! initialize
            lim_vals = 0._dp
            select case(ord_loc)
                case (1:2)
                    max_deriv = 1
                case (3)
                    max_deriv = 3
            end select

            ! interpolation
            call ezspline_interp(f_spl,xb,lim_vals(0),ierr)
            call ezspline_error(ierr)
            chckerr('')

            ! derivatives
            do kd = 1,max_deriv
                call ezspline_derivative(f_spl,kd,xb,lim_vals(kd),ierr)
                call ezspline_error(ierr)
                chckerr('')
            end do
        end function setup_lim_vals

        !> /private calculate extrapolation
        subroutine calc_extrap(xnew,xb,lim_vals,ynew)
```

```
            ! input / output
            real(dp), intent(in) :: xnew(:)                                     ! xnew where to extrapolate
            real(dp), intent(in) :: xb                                          ! x of boundary
            real(dp), intent(in) :: lim_vals(0:2)                               ! limit values
            real(dp), intent(out) :: ynew(:)                                    ! y at xnew

            ! local variables
            real(dp), allocatable :: xdel(:)                                    ! delta x for extrapolation

            allocate(xdel(size(xnew)))
            xdel = xnew-xb

            select case (deriv_loc)
                case (0)
                    ynew = lim_vals(0) + &
                        &lim_vals(1)*xdel + &
                        &lim_vals(2)*xdel**2*0.5_dp
                case (1)
                    ynew = lim_vals(1) + &
                        &lim_vals(2)*xdel
                case (2)
                    ynew = lim_vals(2)
            end select
        end subroutine calc_extrap
    end function spline_real
    !> \private complex version
    integer function spline_complex(x,y,xnew,ynew,ord,deriv,bcs,bcs_val,&
        &extrap) result(ierr)

        use ezspline_obj
        use ezspline

        character(*), parameter :: rout_name = 'spline_complex'

        ! input / output
        real(dp), intent(in) :: x(:)                                            !< coordinates
        complex(dp), intent(in) :: y(:)                                         !< function value
        real(dp), intent(in) :: xnew(:)                                         !< new coordinates
        complex(dp), intent(out) :: ynew(:)                                     !< new function values
        integer, intent(in), optional :: ord                                    !< order [def 3]
        integer, intent(in), optional :: deriv                                  !< derivative [def 0]
        integer, intent(in), optional :: bcs(2)                                 !< boundary conditions [def 0]
        complex(dp), intent(in), optional :: bcs_val(2)                         !< boundary conditions [no def]
        logical, intent(in), optional :: extrap                                 !< whether extrapolation is allowed [def
.false.]

        ! local variables
        real(dp), allocatable :: ynew_loc(:,:)                                  ! local ynew

        ! set up local variables
        allocate(ynew_loc(size(ynew),2))

        ! call real version for real part
        if (present(bcs_val)) then
            ierr = spline_real(x,rp(y),xnew,ynew_loc(:,1),ord,deriv,bcs,&
                &bcs_val=rp(bcs_val),extrap=extrap)
            chckerr('')
        else
            ierr = spline_real(x,rp(y),xnew,ynew_loc(:,1),ord,deriv,bcs,&
                &extrap=extrap)
            chckerr('')
        end if

        ! call real version for complex part
        if (present(bcs_val)) then
            ierr = spline_real(x,ip(y),xnew,ynew_loc(:,2),ord,deriv,bcs,&
                &bcs_val=ip(bcs_val),extrap=extrap)
            chckerr('')
        else
            ierr = spline_real(x,ip(y),xnew,ynew_loc(:,2),ord,deriv,bcs,&
                &extrap=extrap)
            chckerr('')
        end if

        ! save
        ynew = ynew_loc(:,1) + iu*ynew_loc(:,2)
    end function spline_complex

    !> \public  Initialize utilities  for  fast future  reference, depending  on
    !! program style.
    !!
    !! Utilities initialized:
    !!  - derivatives
    !!  - metrics
    !!  - Fourier modes (optionally)
    !!
    !! If Fourier  modes are also initialized,  the quantity \c n_mod  has to be
```

```
    !! provided as well.
    subroutine calc_aux_utilities(n_mod)
        use num_vars, only: max_deriv

        ! input / output
        integer, intent(in), optional :: n_mod                                  !< n_mod for Fourier modes

        ! local variables
        integer :: id, jd, kd                                                   ! counters

        ! common for all program stles

        ! derivatives d from 0 to max_deriv+1
        allocate(d(0:max_deriv+1,0:max_deriv+1,0:max_deriv+1))
        do kd = 0,max_deriv+1
            do jd = 0,max_deriv+1
                do id = 0,max_deriv+1
                    if (id+jd+kd.le.max_deriv+1) then                           ! valid derivatives
                        d(id,jd,kd) = calc_derivs_1d_id([id,jd,kd],3)
                    else                                                        ! derivatives too high
                        d(id,jd,kd) = 0
                    end if
                end do
            end do
        end do

        ! metrics m from 1 to 3
        allocate(m(1:3,1:3))
        do jd = 1,3
            do id = 1,3
                m(id,jd) = calc_derivs_1d_id([id,jd],2)
            end do
        end do

        ! Fourier modes from 1 to n_mod
        if (present(n_mod)) then
            allocate(f(1:n_mod,1:n_mod))
            do jd = 1,n_mod
                do id = 1,n_mod
                    f(id,jd) = calc_derivs_1d_id([id,jd],2)
                end do
            end do
        end if
    end subroutine calc_aux_utilities

    !> Returns the  1D indices  for derivatives  of a  certain order  in certain
    !! dimensions.
    !!
    !! The  algorithm  works by  considering a  structure such  as the  following
    !! example in three dimensions:
    !!  1. (0,0,0)
    !!  2. (1,0,0)
    !!  3. (0,1,0)
    !!  4. (0,0,1)
    !!  5. (2,0,0)
    !!  6. (1,1,0)
    !!  7. (1,0,1)
    !!  8. (0,2,0)
    !!  9. (0,1,1)
    !!  10. (0,0,2)
    !!  11. (3,0,0)
    !!  12. (2,1,0)
    !!
    !! etc...
    !!
    !! By then  defining \f$d\f$ as  the vector  of derivatives, \f$n\f$  as the
    !! size  of  \f$d\f$, \f$D  =  \sum_i^n  d(i)\f$  the  total degree  of  the
    !! derivative,  and \f$I\f$  as the  index of  the last  nonzero element  in
    !! \f$d\f$, and extending \f$d\f$ to the left by considering \f$d(0)\f$ = 0,
    !! the following formula can be can be deduced for the displacements in this
    !! table with respect to index 1:
    !!
    !! displacements in this table with respect to index 1:
    !!  \f[\sum_{j=0}^{I-1} \sum_{i=0}^
    !!      {D-(d(0)+...+d(j))-1}
    !!      \left(\begin{array}{c}n-j+i-1\\i\end{array}\right) ,
    !!  \f]
    !! making use of the binomial coefficients
    !!  \f[\left(\begin{array}{c}a\\b\end{array}\right) = \frac{a!}{b!(a-b)!}\f]
    !!
    !! It can be
    !! seen that  each of the terms  in the summation in  \f$j\f$ corresponds to
    !! the displacement in dimension \f$j\f$  and the binomial coefficient comes
    !! from the classic stars and bars problem.
    integer function calc_derivs_1d_id(deriv,dims) result(res)
        ! input / output
        integer, intent(in) :: deriv(:)                                         !< derivatives
```

```
        integer, intent(in) :: dims                                     !< nr. of dimensions

        ! local variables
        integer :: id, jd                                               ! counters
        integer :: id_max                                               ! index of last nonzero element in deriv
        integer :: tot_deriv                                            ! total derivative
        integer, allocatable :: deriv_loc(:)                            ! local extended deriv

        ! set up local deriv
        allocate(deriv_loc(0:size(deriv)))
        deriv_loc(0) = 0
        deriv_loc(1:size(deriv)) = deriv

        ! set up total derivative
        tot_deriv = sum(deriv)

        ! initialize res
        res = 1

        ! calculate  displacement  if total  deriv not  equal to  zero (assuming
        ! deriv > 0)
        if (tot_deriv.gt.0) then
            ! get index of last nonzero element in deriv
            id_max = 0
            do id = 1,size(deriv)
                if (deriv(id).gt.0) id_max = id
            end do

            ! iterate over dimensions
            do jd = 0,id_max-1
                ! iterate over derivatives of dimension jd (where deriv(jd) = 0)
                do id = 0,tot_deriv-sum(deriv_loc(0:jd))-1
                    res = res + fac(dims-jd+id-1)/(fac(id)*fac(dims-jd-1))
                end do
            end do
        end if
    end function calc_derivs_1d_id

    !> Calculate  all of  the  unique  derivatives in  certain  dimensions of  a
    !! certain total order.
    !!
    !! For example, for order 2 in 3 dimensions this would be:
    !!  - (2,0,0)
    !!  - (1,1,0)
    !!  - (1,0,1)
    !!  - (0,2,0)
    !!  - (0,1,1) and
    !!  - (0,0,2)
    !!
    !! The total  number of  derivatives  is found  through the  stars and  bars
    !! problem to be the binomial coefficient
    !!   \f[\left(\begin{array}{c}order+dims-1\\order\end{array}\right) =
    !!      \frac{(order+dims-1)!}{order!(dims-1)!} . \f]
    !! The number  of dimensions is taken to  be 3 by default but  can be changed
    !! optionally.
    recursive subroutine calc_derivs_of_order(res,order,dims)
        ! input / output
        integer, intent(inout), allocatable :: res(:,:)                 !< array of all unique derivatives
        integer, intent(in) :: order                                    !< order of derivative
        integer, intent(in), optional :: dims                           !< nr. of dimensions

        ! local variables
        integer :: id                                                   ! counter
        integer, allocatable :: derivs_loc(:,:)                         ! derivs of local subblock
        integer :: id_tot                                               ! counter for total dimension of local derivs.
        integer :: dims_loc                                             ! local version of dims

        ! set up local dims
        dims_loc = 3
        if (present(dims)) dims_loc = dims

        ! allocate resulting derivs
        if (allocated(res)) deallocate(res)
        allocate(res(&
            &dims_loc,fac(order+dims_loc-1)/(fac(order)*fac(dims_loc-1))))

        ! iterate over the first index if order greater than 0
        if (order.gt.0) then
            id_tot = 0
            do id = order,0,-1
                ! calculate  the  derivs  of  local  subblock  if  more  than  1
                ! dimension
                if (dims_loc.gt.1) then
                    call calc_derivs_of_order(derivs_loc,order-id,dims_loc-1)   ! calculate iteratively subblock
                    res(1,id_tot+1:id_tot+size(derivs_loc,2)) = id              ! set first dimension
                    res(2:dims_loc,id_tot+1:id_tot+size(derivs_loc,2)) &
```

```
                    &= derivs_loc                                       ! set next dimensions
                id_tot = id_tot+size(derivs_loc,2)
            else
                res = order
            end if
        end do
    else
        res = 0
    end if
end subroutine calc_derivs_of_order
!> Returns derivatives of certain order.
function derivs(order,dims) result(res)
    ! input / output
    integer, intent(in) :: order                                        !< order of derivative
    integer, intent(in), optional :: dims                               !< nr. of dimensions
    integer, allocatable :: res(:,:)                                    !< array of all unique derivatives

    ! call the subroutine
    call calc_derivs_of_order(res,order,dims)
end function derivs

!> checks whether  the derivatives  requested for  a certain  subroutine are
!! valid
!!
!! \return ierr
integer function check_deriv(deriv,max_deriv,sr_name) result(ierr)
    character(*), parameter :: rout_name = 'check_deriv'

    ! input / output
    integer, intent(in) :: deriv(3)                                     !< derivative to be checked
    integer, intent(in) :: max_deriv                                    !< maximum derivative allowed
    character(len=*), intent(in) :: sr_name                             !< name of subroutine

    ! local variables
    integer :: id
    character(len=max_str_ln) :: err_msg                                ! error message

    ! initialize ierr
    ierr = 0

    ! test the derivatives
    do id = 1, 3
        if (deriv(id).gt.max_deriv) then
            err_msg = 'Asking '//trim(sr_name)//' for a derivative&
                & in the '//trim(i2str(id))//'th dimension of order '&
                &//trim(i2str(deriv(id)))//', while the maximum order is '&
                &//trim(i2str(max_deriv))
            ierr = 1
            chckerr(err_msg)
        end if
    end do
end function

!> Extrapolates a function.
!!
!! This is  done using linear  or quadratic interpolation, depending  on the
!! number of points and values given.
!!
!! The data  should be given  sorted, in ascending order,  without duplicate
!! points in var_points.
!!
!! It uses the following solution:
!!  \f[\overline{\text{A}} \ \vec{b} = \vec{c}\f]
!! which is written out to
!!  \f[\left(\begin{array}{cccc}
!!          x_1^0  & x_1^1  & x_1^2  & \ldots \\
!!          x_2^0  & x_2^1  & x_2^2  & \ldots \\
!!          x_3^0  & x_3^1  & x_3^2  & \ldots \\
!!          \vdots & \vdots & \vdots & \ddots \\
!!      \end{array}\right)
!!      \left(\begin{array}{c}
!!          a_0 \\ a_1 \\ a_2 \\ \vdots
!!      \end{array}\right)
!!      =
!!      \left(\begin{array}{c}
!!          v_0 \\ v_1 \\ v_2 \\ \vdots
!!      \end{array}\right)
!!  \f]
!! where \f$v_{i-1}\f$ = \c var_i.
!!
!! This is solved for the coefficients of the interpolating polynomial
!!
!!  <tt>ext_var = a_0 + a_1 x + a_2 x^2 + ...</tt>
!!
!! There is an optional flag to look for the \f$k^\text{th}\f$ derivative of
!! the  function instead  of the  function itself,  where \f$k\f$  should be
!! lower than the degree of the polynomial.
```

```
    !!
    !! \return ierr
    integer function calc_ext_var(ext_var,var,var_points,ext_point,deriv_in) &
        &result(ierr)
        character(*), parameter :: rout_name = 'ext_var'

        ! input / output
        real(dp), intent(inout) :: ext_var                                      !< output
        real(dp), intent(in) :: var(:)                                          !< abscissa
        real(dp), intent(in) :: var_points(:)                                   !< ordinate
        real(dp), intent(in) :: ext_point                                       !< point to which extrapolate
        integer, intent(in), optional :: deriv_in                               !< specifies an optional derivative

        ! local variables
        integer :: id, jd
        integer :: istat
        integer :: pol_deg
        integer :: deriv
        real(dp), allocatable :: a(:,:), lu(:,:)
        real(dp), allocatable :: a_i(:)
        real(dp) :: fact                                                        ! multiplicative factor, see below
        character(len=max_str_ln) :: err_msg                                    ! error message

        ! initialize ierr
        ierr = 0

        ! determine the degree of the polynomial ext_var
        pol_deg = size(var)

        ! tests
        if (size(var).ne.size(var_points)) then
            err_msg = 'The size of the arrays containing the variable and &
                &the points do not match'
            ierr = 1
            chckerr(err_msg)
        end if
        if (present(deriv_in)) then
            deriv = deriv_in
            if (deriv.ge.pol_deg) then                                          ! order of derivative too hight
                err_msg = 'Asking ext_var for '//trim(i2str(deriv_in))&
                    &//'th derivative, while using a polynomial of degree '&
                    &//trim(i2str(pol_deg))
                ierr = 1
                chckerr(err_msg)
            else if (deriv.lt.0) then
                err_msg = 'Asking ext_var for a derivative of order '&
                    &//trim(i2str(deriv_in))
                ierr = 1
                chckerr(err_msg)
            end if
        else
            deriv = 0
        end if

        ! fill matrix
        allocate(a(pol_deg,pol_deg))
        allocate(lu(pol_deg,pol_deg))
        a(:,1) = 1.0_dp
        col: do jd = 2, pol_deg
            row: do id = 1, pol_deg
                a(id,jd) = var_points(id)**(jd-1)
            end do row
        end do col

        ! solve the system for a_i
        allocate(a_i(pol_deg))
        a_i = var
        lu = a
        call dgesv( pol_deg, 1, lu, pol_deg, [(id,id=1,pol_deg)], &
            &a_i, pol_deg, istat)


        if (istat.ne.0) then
            err_msg = 'The solution could not be found. Are the extrapolating &
                &points independent?'
            ierr = 1
            if (ierr.ne.0) then
                call writo('The matrix equation to be solved was Ax=b, A = ')
                call print_ar_2(a)
                call writo('and b = ')
                call print_ar_1(var)
            end if
            chckerr(err_msg)
        end if

        ext_var = 0.0_dp
        ! for deriv of order i:
```

```
        ! d^i/dx^i f = sum_j = i ^ pol_deg ( j*(j-1)*...*(j-i) a_j x^(j-i) )
        polynom: do id = 1+deriv, pol_deg
            ! construct the factor
            fact = 1.0_dp
            d_dx: do jd = 1,deriv
                fact = fact * float((id-1)-(jd-1))
            end do d_dx
            ! update ext_var with current polynomial term a_i(id)
            ext_var = ext_var + fact*a_i(id)*ext_point**(id-1-deriv)
        end do polynom
    end function calc_ext_var

    !> Calculate the coefficients for finite differences.
    !!
    !! These represent a  derivative of degree \c  deriv at an index  \c ind, by
    !! the weighted sum of a number \c nr of points with <tt>1<=ind<=nr</tt>.
    !!
    !! They are found by solving a Vandermonde system, multiplied by the faculty
    !! of the derivative.
    !!
    !! The result  returned in <tt>coeff(1:nr)</tt> are used in
    !!  \f[\frac{d f}{d x} =
    !!      \sum_{j=1}^n c(j) f(i+j) . \f]
    !!
    !! where \f$i\f$ = \c ind, \f$n\f$ = \c nr and \f$c\f$ = \c coeff
    !!
    !! \note They need to be divided by the step size before usage.
    !!
    !! Examples:
    !!  - symmetric finite differences for derivative \c deriv of order \c ord:
    !!      - \f$m = \text{ceiling}(\frac{p+d}{2})\f$
    !!          to guarantee the order
    !!      - \f$n   = 1+2m\f$
    !!      - \f$i  = 1+m\f$
    !!  - left finite differences for derivative \c deriv of order \c ord:
    !!      - \f$n   = 1+d+p \f$
    !!      - \f$i  = m\f$
    !! where \f$p\f$ = \c ord, \f$d\f$ = \c deriv and \f$m\f$ = \c nr_2.
    !!
    !! \return ierr
    integer function calc_coeff_fin_diff(deriv,nr,ind,coeff) result(ierr)
        character(*), parameter :: rout_name = 'calc_coeff_fin_diff'

        ! input / output
        integer, intent(in) :: deriv                                        !< degree of derivative
        integer, intent(in) :: nr                                           !< number of points
        integer, intent(in) :: ind                                          !< position of derivative
        real(dp), intent(inout), allocatable :: coeff(:)                    !< output coefficients

        ! local variables
        character(len=max_str_ln) :: err_msg                                ! error message
        real(dp), allocatable :: mat_loc(:)                                 ! elements of local Vandermonde matrix
        real(dp), allocatable :: rhs_loc(:)                                 ! local right-hand side
        integer :: id                                                       ! counter

        ! initialize ierr
        ierr = 0

        ! test  whether  number of  points  nr  and  position ind  positive  and
        ! consistent
        if (deriv.le.0) then
            ierr = 1
            err_msg = 'The degree of the derivative has to be positive'
            chckerr(err_msg)
        end if
        if (nr.le.0) then
            ierr = 1
            err_msg = 'The number of points in the finite differences has to &
                &be positive'
            chckerr(err_msg)
        end if
        if (ind.lt.1 .or. ind.gt.nr) then
            ierr = 1
            err_msg = 'The position of the derivative is not in range.'
            chckerr(err_msg)
        end if

        ! test whether number of points enough for derivative
        if (deriv.gt.nr-1) then
            ierr = 1
            err_msg = 'For derivatives of degree '//trim(i2str(deriv))//&
                &' minimally '//trim(i2str(deriv+1))//' points are necessary'
            chckerr(err_msg)
        end if

        ! set up output
        if (allocated(coeff)) deallocate(coeff)
```

```fortran
        allocate(coeff(nr))
        allocate(mat_loc(nr))
        allocate(rhs_loc(nr))

        ! set up matrix and rhs
        do id = 1,nr
            mat_loc(id) = (id-ind)*1._dp
        end do
        rhs_loc = 0._dp
        rhs_loc(deriv+1) = 1._dp                                        ! looking for this derivative
        call solve_vand(nr,mat_loc,rhs_loc,coeff,transp=.true.)
        coeff = coeff*fac(deriv)
    end function calc_coeff_fin_diff

    !> Convert 2-D coordinates (i,j) to the storage convention used in matrices.
    !!
    !! Their size is by default taken to be 3:
    !!  \f[
    !!      \left(\begin{array}{ccc}
    !!          1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9
    !!      \end{array}\right)
    !!      \ \text{or} \
    !!      \left(\begin{array}{ccc}
    !!          1 &   &   \\ 2 & 4 &   \\ 3 & 5 & 6
    !!      \end{array}\right) \
    !!      \text{for symmetic matrices} \ .
    !!  \f]
    !! Optionally, this can be changed using \c n.
    !!
    !! The value of \f$c\f$ is then given by
    !!   \f[c = (j-1) n + i\f]
    !! for non-symmetric matrices, and by
    !!  \f[\begin{aligned}
    !!      c &= (j-1)n + i - (j-1)\frac{j}{2} &  \text{if} \ i > j \\
    !!      c &= (i-1)n + j - (i-1)\frac{i}{2} &  \text{if} \ j > i
    !!  \end{aligned}\f]
    !! since \f$\sum_{k=1}^{j-1} k = \frac{(j-1)j}{2}\f$.
    !!
    !! For local indices  in submatrices, the limits in both  dimensions have to
    !! be passed. The results for the  full matrix are then subtracted by amount
    !! corresponding to  the left,  above and  below parts  with respect  to the
    !! submatrix:
    !!  - left:
    !!      \f[\sum_{i=1}^{m(2)-1} n-i+1 =
    !!          (m(2)-1) \left(n+1-\frac{m(2)}{2}\right) \f]
    !!  - above (if positive):
    !!      \f[\sum_{i=1}^j \left(m(1)-m(2)+1-i\right) =
    !!          j \left(m(1)-m(2)+\frac{1}{2} -
    !!          \frac{j^*}{2}\right) \ , \ \text{with} \ j^* =
    !!          \text{min}\left(0,j,m(1)-m(2)+1\right) \f]
    !!  - below:
    !!      \f[\sum_{i=1}^{j-1} n-M(1) =
    !!          \left(n-M(1)\right) \left(j-1\right) \f]
    !!
    !! where \f$m\f$ = \c min and \f$M\f$ = \c max.
    !!
    !! \note
    !!  -# The  submatrix version is  not fast, so  results should be  saved and
    !!  reused.
    !!  -# No checks are done whether the indices make sense.
    integer function c(ij,sym,n,lim_n)
        ! input / output
        integer, intent(in) :: ij(2)                                    !< 2D coords. (i,j)
        logical, intent(in) :: sym                                      !< whether symmetric
        integer, intent(in), optional :: n                              !< max of 2-D coords.
        integer, intent(in), optional :: lim_n(2,2)                     !< min. and max. of 2-D coords.

        ! local variables
        integer :: n_loc                                                ! local n
        integer :: js                                                   ! j* = min(0,j,min(1)-min(2)+1)
        integer :: ij_loc(2)                                            ! local ij
        integer :: kd                                                   ! dummy integer

        ! set local n
        n_loc = 3
        if (present(n)) n_loc = n

        ! set c depending on symmetry
        if (sym) then                                                   ! symmetric
            ! set local ij, refering to the total indices
            ij_loc = ij
            if (present(lim_n)) ij_loc = ij + lim_n(1,:) - 1            ! displace with minimum indices

            ! swap indices if upper diagonal
            if (ij_loc(2).gt.ij_loc(1)) then
                kd = ij_loc(2)
                ij_loc(2) = ij_loc(1)
```

```
                ij_loc(1) = kd
            end if

            ! get c for whole matrix
            c = nint((ij_loc(2)-1)*n_loc + ij_loc(1) - &
                &(ij_loc(2)-1)*ij_loc(2)*0.5)

            if (present(lim_n)) then
                ! subtract if submatrix
                js = max(0,min(ij_loc(2)-lim_n(1,2)+1,lim_n(1,1)-lim_n(1,2)+1))
                c = c - nint((lim_n(1,2)-1)*(n_loc+1-lim_n(1,2)*0.5) &
                    &+js*(lim_n(1,1)-lim_n(1,2)+0.5-js*0.5) &
                    &+(n_loc-lim_n(2,1))*(ij_loc(2)-lim_n(1,2)))
            end if
        else                                                            ! asymmetric
            if (present(lim_n)) then
                ! set c with local size
                c = (ij(2)-1)*(lim_n(2,1)-lim_n(1,1)+1) + ij(1)
            else
                ! set c with total size
                c = (ij(2)-1)*n_loc + ij(1)
            end if
        end if
    end function c

    !> Calculate factorial.
    recursive function fac(n)  result(fact)
        ! input / output
        integer, intent(in) :: n                                        !< input
        integer :: fact                                                 !< output

        ! calculate recursively
        if (n.eq.0) then
            fact = 1
        else
            fact = n * fac(n-1)
        end if
    end function fac

    !> Determines  whether a  matrix making  use  of the  storage convention  in
    !! eq_vars.eq_2_type is symmetric or not.
    !!
    !! \return ierr
    integer function is_sym(n,nn,sym) result(ierr)
        character(*), parameter :: rout_name = 'is_sym'

        ! input / output
        integer, intent(in) :: n                                        !< size of matrix
        integer, intent(in) :: nn                                       !< number of elements in matrix
        logical, intent(inout) :: sym                                   !< output

        ! local variables
        character(len=max_str_ln) :: err_msg                            ! error message

        ! initialize ierr
        ierr = 0

        ! set sym
        if (nn.eq.n**2) then
            sym = .false.
        else if (nn.eq.n*(n+1)/2) then
            sym = .true.
        else
            ierr = 1
            err_msg = 'The matrix corresponds neither to a symmetric nor to a &
                &normal matrix'
            chckerr('')
        end if
    end function is_sym

    !> Returns common multiple using the Euclid's algorithm.
    !!
    !! \see From
    !!  <https://rosettacode.org/wiki/Greatest_common_divisor#Recursive_Euclid_algorithm_3>
    recursive function lcm(u, v) result(res)
        integer, intent(in) :: u                                        !< input
        integer, intent(in) :: v                                        !< input
        integer :: res                                                  !< result

        res = u*v / gcd(u,v)
    end function lcm

    !> Returns least denominator using the GCD.
    !!
    !! \see From
    !!  <https://rosettacode.org/wiki/Least_common_multiple#Fortran>
    recursive function gcd(u, v) result(res)
```

```
        integer, intent(in) :: u                                        !< input
        integer, intent(in) :: v                                        !< input
        integer :: res                                                  !< result

        if (mod(u, v) /= 0) then
            res = gcd(v, mod(u, v))
        else
            res = v
        end if
    end function gcd

    !> Calculate multiplication  through shifting of fourier modes A  and B into
    !! C.
    !!
    !! This works by calculating
    !! \f$\left[\sum_A \left(\alpha_A \cos m_A \theta + \beta_A \sin m_A \theta \right)\right]
    !! \left[\sum_B \left(\alpha_B \cos m_B \theta + \beta_B \sin m_B \theta \right)\right] =
    !! \left[\sum_C \left(\alpha_C \cos m_C \theta + \beta_C \sin m_C \theta \right)\right]\f$
    !!
    !! where the \f$\alpha\f$ and \f$\beta\f$ factors  are provide in \c A, \c B
    !! and \c C.
    !!
    !! This  then boils  down to  finding the four  combinations of  cosines and
    !! sines.
    !!
    !! \note Modes that are larger than what C can hold are thrown away.
    subroutine shift_f(Al,Bl,Cl,A,B,C)
        ! input / output
        integer, intent(in) :: al(2)                                    !< limits on A mode numbers
        integer, intent(in) :: bl(2)                                    !< limits on B mode numbers
        integer, intent(in) :: cl(2)                                    !< limits on C mode numbers
        real(dp), intent(in) :: a(al(1):al(2),2)                        !< input
        real(dp), intent(in) :: b(bl(1):bl(2),2)                        !< input
        real(dp), intent(inout) :: c(cl(1):cl(2),2)                     !< result

        ! local variables
        integer :: i_a, i_b, i_c                                        ! indices in A, B and C

        ! initialize C
        c = 0._dp

        ! loop over A
        do i_a = al(1),al(2)
            do i_b = bl(1),bl(2)
                ! contribution to i_A + i_B
                i_c = i_a + i_b

                if (i_c.ge.cl(1) .and. i_c.le.cl(2)) then
                    c(i_c,1) = c(i_c,1) + 0.5* a(i_a,1)*b(i_b,1)
                    c(i_c,2) = c(i_c,2) + 0.5* a(i_a,1)*b(i_b,2)
                    c(i_c,2) = c(i_c,2) + 0.5* a(i_a,2)*b(i_b,1)
                    c(i_c,1) = c(i_c,1) - 0.5* a(i_a,2)*b(i_b,2)
                end if

                ! contribution to i_A + i_B
                i_c = i_a - i_b

                if (i_c.ge.cl(1) .and. i_c.le.cl(2)) then
                    c(i_c,1) = c(i_c,1) + 0.5* a(i_a,1)*b(i_b,1)
                    c(i_c,2) = c(i_c,2) - 0.5* a(i_a,1)*b(i_b,2)
                    c(i_c,2) = c(i_c,2) + 0.5* a(i_a,2)*b(i_b,1)
                    c(i_c,1) = c(i_c,1) + 0.5* a(i_a,2)*b(i_b,2)
                end if
            end do
        end do
    end subroutine shift_f

    !> Solve a Vandermonde system \f$\overline{\text{A}} \ \vec{X} = \vec{B}\f$.
    !!
    !! The Vandermonde matrix has the form
    !!  \f[A = \left(\begin{array}{ccccc}
    !!      1      & a_1    & a_1^2  & \cdots & a_1^n \\
    !!      1      & a_2    & a_2^2  & \cdots & a_2^n \\
    !!      1      & a_3    & a_3^2  & \cdots & a_3^n \\
    !!      \vdots & \vdots & \vdots & \ddots & \vdots \\
    !!      1      & a_n    & a_n^2  & \cdots & a_n^n
    !!  \end{array}\right)\f]
    !!
    !! \see Adapted from two routines \c dvand and \c pvand in
    !! <https://people.sc.fsu.edu/~jburkardt/f_src/vandermonde/vandermonde.html>
    !! based on the Björk-Pereyra Algorithm \cite Bjorck1970.
    subroutine solve_vand(n,a,b,x,transp)
        ! input / output
        integer, intent(in) :: n                                        !< matrix size
        real(dp), intent(in) :: a(n)                                    !< parameters of Vandermonde matrix
        real(dp), intent(in) :: b(n)                                    !< right-hand side
        real(dp), intent(inout) :: x(n)                                 !< solution
```

```
        logical, intent(in), optional :: transp                                 !< transposed Vandermonde matrix

        ! local variables
        integer :: jd, kd                                                       ! counters
        logical :: transp_loc                                                   ! local transp

        ! initialize
        transp_loc = .false.
        if (present(transp)) transp_loc = transp
        x(1:n) = b(1:n)

        if (transp_loc) then
            do kd = 1, n - 1
                do jd = n, kd + 1, -1
                    x(jd) = x(jd) - a(kd) * x(jd-1)
                end do
            end do

            do kd = n - 1, 1, -1
                do jd = kd + 1, n
                    x(jd) = x(jd) / ( a(jd) - a(jd-kd) )
                end do
                do jd = kd, n - 1
                    x(jd) = x(jd) - x(jd+1)
                end do
            end do
        else
            do kd = 1, n - 1
                do jd = n, kd + 1, -1
                    x(jd) = ( x(jd) - x(jd-1) ) / ( a(jd) - a(jd-kd) )
                end do
            end do

            do kd = n - 1, 1, -1
                do jd = kd, n - 1
                    x(jd) = x(jd) - a(kd) * x(jd+1)
                end do
            end do
        end if
    end subroutine solve_vand

#if ldebug
    !> \public   Calculate  second   derivative   with   smoothing  formula   by
    !! Holoborodko, \cite holoborodko2008diff
    !!
    !! The formula used is
    !!
    !! \f$s\left(k\right) = \left[\left(2N-10\right)s\left(k+1\right) -
    !!  \left(N+2k+3\right)s\left(k+2\right)\right]/\left(N-2k-1\right)\f$
    !!
    !! with \f$s\left(k>M\right) = 0\f$ and \f$\left(k=N\right) = 1\f$.
    !!
    !! for a given \f$N\f$.
    !!
    !! For \c style 1, central differences are used:
    !!
    !! \f$\frac{\partial^2 y}{\partial x^2} \approx \frac{1}{2^{N-3}}
    !!  \left(\sum_{k=1}^M \alpha_k \left(y_k + y_{-k} \right) -
    !!  2 y_0 \sum_{k=1}^M \alpha_k \right)\f$
    !!
    !! and for \c style 2, backward differences:
    !!
    !! \f$\frac{\partial^2 y}{\partial x^2} \approx \frac{1}{2^{N-3}}
    !!  \left(\sum_{k=1}^M \beta_k \left(y_{-M+k} + y_{-M-k} \right) -
    !!  2 y_{-M} \sum_{k=1}^M \beta_k \right)\f$
    !!
    !! with \f$\alpha_k\f$ and \f$\beta_k\f$ given by
    !!
    !! \f$\begin{aligned}
    !!  \alpha_k &= \frac{4k^2 s_k}{\left(x_k-x_{-k}\right)^2} \\
    !!  \beta_k &= \frac{4k^2 s_k}{\left(x_{-M+k}-x_{-M-k}\right)^2}
    !! \end{aligned}\f$.
    integer function calc_d2_smooth(fil_N,x,y,D2y,style) result(ierr)
        character(*), parameter :: rout_name = 'calc_D2_smooth'

        ! input / output
        integer, intent(in) :: fil_n                                            !< filter length (must be odd)
        real(dp), intent(in) :: x(:)                                            !< input abscissa
        real(dp), intent(in) :: y(:)                                            !< input ordinate
        real(dp), intent(inout) :: d2y(:)                                       !< second derivative
        integer, intent(in) :: style

        ! local variables
        integer :: n                                                            ! size of arrays
        integer :: id, kd                                                       ! counters
        integer :: sym_m                                                        ! center of symmetry
        real(dp), allocatable :: s(:)                                           ! coeffcients s
```

```
        real(dp), allocatable :: ab(:)                                  ! alpha (style 1) or beta (style 2)
        character(len=max_str_ln) :: err_msg                            ! error message

        ! initialize ierr
        ierr = 0

        ! tests
        if (fil_n.lt.5) then
            ierr = 1
            err_msg = 'filter length must be at least 5'
            chckerr(err_msg)
        end if
        if (mod(fil_n,2).ne.1) then
            ierr = 1
            err_msg = 'filter length must be odd'
            chckerr(err_msg)
        end if

        ! set size of arrays
        sym_m = (fil_n-1)/2
        n = size(x)
        allocate(s(0:sym_m+1))
        allocate(ab(sym_m))

        ! calculate coefficients s
        s(sym_m+1) = 0._dp
        s(sym_m) = 1._dp
        do kd = sym_m-1,0,-1
            s(kd) = ((2*fil_n-10)*s(kd+1) - (fil_n+2*kd+3)*s(kd+2)) / &
                &(fil_n-2*kd-1)
        end do

        ! calculate for all n-fil_N interior (style 1) or left (style 2) points
        ! The second derivative at other points are set to zero.
        select case (style)
            case (1)                                                    ! central
                d2y(1:sym_m) = 0._dp
                d2y(n-sym_m+1:n) = 0._dp
                do id = sym_m+1,n-sym_m
                    ! calculate alpha
                    do kd = 1,sym_m
                        ab(kd) = 4*kd**2*s(kd)*(x(id+kd)-x(id-kd))**(-2)
                    end do
                    ! set D2y
                    d2y(id) = sum(ab*(y(id+1:id+sym_m))) + &
                        &sum(ab*y(id-1:id-sym_m:-1)) - &
                        &2*y(id)*sum(ab)
                end do
            case (2)                                                    ! left
                d2y(1:2*sym_m) = 0._dp
                do id = 2*sym_m+1,n
                    ! calculate beta
                    do kd = 1,sym_m
                        ab(kd) = 4*kd**2*s(kd)*(x(id+kd-sym_m)-x(id-kd-sym_m))**(-2)
                    end do
                    ! set D2y
                    d2y(id) = sum(ab*(y(id+1-sym_m:id))) + &
                        &sum(ab*y(id-1-sym_m:id-2*sym_m:-1)) - &
                        &2*y(id-sym_m)*sum(ab)
                end do
            case default
                err_msg = 'No style associated with '//trim(i2str(style))
                ierr = 1
                chckerr(err_msg)
        end select

        ! rescale
        d2y = d2y * 2._dp**(3._dp-fil_n)
    end function calc_d2_smooth
#endif
end module num_utilities
```

D.60 /Users/toonweyens/Code/PB3D/Modules/num_vars.f90 File Reference

Modules

- module num_vars

  Numerical variables used by most other modules.

Variables

- integer, parameter, public num_vars::dp = REAL64
  double precision
- integer, parameter, public num_vars::dpi = INT64
  double precision
- real(dp), parameter, public num_vars::weight_dp = 0.008
  size of double precision in kB
- integer, parameter, public num_vars::max_str_ln = 120
  maximum length of strings
- integer, parameter, public num_vars::max_name_ln = 30
  maximum length of filenames
- integer, parameter, public num_vars::max_deriv = 2
  highest derivatives for metric factors in Flux coords.
- integer, public num_vars::prog_style
  program style (1: PB3D, 2: PB3D_POST)
- character(len=4), public num_vars::prog_name
  name of program, used for info
- character(len=3), parameter, public num_vars::output_name = 'out'
  name of output file
- character(len=14), parameter, public num_vars::shell_commands_name = 'shell_commands'
  name of shell commands file
- character(len=9), parameter, public num_vars::mem_usage_name = 'mem_usage'
  name of memory usage file
- integer, public num_vars::mem_usage_count
  counter for memory usage output
- real(dp), parameter, public num_vars::prog_version = 2.47_dp
  version number
- real(dp), parameter, public num_vars::min_pb3d_version = 2.43_dp
  minimum PB3D version for POST
- logical, public num_vars::debug_version = .true.
  debug version used
- integer, public num_vars::rank
  MPI rank.
- integer, public num_vars::n_procs
  nr. of MPI processes
- integer, public num_vars::sol_n_procs
  nr. of MPI processes for solution with SLEPC
- integer(kind=8), public num_vars::time_start
  start time of simulation
- real(dp), public num_vars::max_tot_mem
  maximum total memory for all processes [MB]
- real(dp), public num_vars::max_x_mem
  maximum memory for perturbation calculations for all processes [MB]
- integer, dimension(:,:), allocatable, public num_vars::x_jobs_lims
  data about X jobs: [$\min_k$, $\max_k$, $\min_m$, $\max_m$ *] for all jobs*
- integer, dimension(:,:), allocatable, public num_vars::eq_jobs_lims
  data about eq jobs: [$\min_\theta$, $\max_\theta$ *] for all jobs*
- integer, public num_vars::x_job_nr
  nr. of X job
- integer, public num_vars::eq_job_nr

nr. of eq job

- real(dp), parameter, public num_vars::mem_scale_fac = 6.0

  empirical scale factor of memory to calculate eq compared to just storing it
- real(dp), parameter, public num_vars::pi =4_dp∗datan(1.0_dp)

  π
- real(dp), parameter, public num_vars::mu_0_original = 4E-7_dp∗pi

  permeability of free space
- complex(dp), parameter, public num_vars::iu = (0, 1)

  complex unit
- integer, public num_vars::ev_style

  determines the method used for solving an EV problem
- integer, public num_vars::eq_style

  either 1 (VMEC) or 2 (HELENA)
- integer, public num_vars::rho_style

  style for equilibrium density profile
- integer, public num_vars::u_style

  style for calculation of U (1: ord.2, 2: ord.1, 1: ord.0)
- integer, public num_vars::norm_style

  style for normalization
- integer, public num_vars::k_style

  style for kinetic energy
- integer, dimension(2), public num_vars::bc_style

  style for BC left and right
- integer, public num_vars::x_style

  style for secondary mode numbers (1: prescribed, 2: fast)
- integer, public num_vars::matrix_slepc_style

  style for matrix storage (1: sparse, 2: shell)
- integer, public num_vars::solver_slepc_style

  style for solver (1: Krylov-Schur, 2: GD)
- integer, public num_vars::post_style

  style for POST (1: extended grid, 2: B-aligned grid)
- integer, public num_vars::x_grid_style

  style for normal component of X grid (1: eq, 2: sol, 3: enriched)
- integer, public num_vars::alpha_style

  style for alpha (1: one field line, many turns, 2: many field lines, one turn)
- integer, public num_vars::max_it_slepc

  maximum nr. of iterations for SLEPC
- logical, public num_vars::plot_resonance

  whether to plot the q-profile or iota-profile with resonances
- logical, public num_vars::plot_magn_grid

  whether to plot the grid in real coordinates
- logical, public num_vars::plot_b

  whether to plot the magnetic field in real coordinates
- logical, public num_vars::plot_j

  whether to plot the current in real coordinates
- logical, public num_vars::plot_flux_q

  whether to plot flux quantities in real coordinates
- logical, public num_vars::plot_kappa

  whether to plot curvature
- logical, public num_vars::plot_sol_xi

  whether to plot plasma perturbation of solution in POST

- logical, public num_vars::plot_sol_q

  whether to plot magnetic perturbation of solution in POST
- logical, public num_vars::plot_vac_pot

  whether to plot vacuum potential in POST
- logical, public num_vars::plot_e_rec

  whether to plot energy reconstruction in POST
- logical, public num_vars::ltest

  whether or not to call the testing routines
- logical, public num_vars::use_pol_flux_e

  whether poloidal flux is used in E coords.
- logical, public num_vars::use_pol_flux_f

  whether poloidal flux is used in F coords.
- logical, public num_vars::use_normalization

  whether to use normalization or not
- real(dp), public num_vars::ev_bc

  value of artificial Eigenvalue for boundary condition
- real(dp), public num_vars::ev_guess

  first guess for eigenvalue
- real(dp), dimension(:), allocatable, public num_vars::tol_slepc

  tolerance for SLEPC for different Richardson levels
- real(dp), public num_vars::max_njq_change

  maximum change of prim. mode number times saf. fac. / rot. transf. when using X_style 2 (fast)
- integer, public num_vars::norm_disc_prec_eq

  precision for normal discretization for equilibrium
- integer, public num_vars::norm_disc_prec_x

  precision for normal discretization for perturbation
- integer, public num_vars::norm_disc_prec_sol

  precision for normal discretization for solution
- integer, public num_vars::norm_disc_style_sol

  style for normal discretization for solution (1: central fin. diff., 2: left fin. diff.)
- integer, public num_vars::magn_int_style

  style for magnetic integrals (1: trapezoidal, 2: Simpson 3/8)
- integer, public num_vars::max_it_rich

  number of levels for Richardson extrapolation
- real(dp), public num_vars::tol_rich

  tolerance for Richardson extrapolation
- integer, public num_vars::max_it_zero

  maximum number of iterations to find zeros
- integer, public num_vars::max_nr_backtracks_hh

  maximum number of backtracks for Householder, relax. factors
- real(dp), public num_vars::tol_zero

  tolerance for zeros
- real(dp), public num_vars::tol_norm

  tolerance for normal range (normalized to 0..1)
- integer, parameter, public num_vars::ex_max_size = 10000

  maximum size of matrices for external plot
- character(len=max_str_ln), public num_vars::eq_name

  name of equilibrium file from VMEC or HELENA
- character(len=max_str_ln), public num_vars::pb3d_name

  name of PB3D output file
- logical, public num_vars::no_plots = .false.

no plots made

- logical, public num_vars::jump_to_sol = .false.

  jump to solution
- logical, public num_vars::export_hel = .false.

  export HELENA
- logical, public num_vars::plot_vmec_modes = .false.

  plot VMEC modes
- logical, public num_vars::invert_top_bottom_h = .false.

  invert top and bottom for HELENA equilibria
- logical, public num_vars::no_output = .false.

  no output shown
- logical, public num_vars::post_output_full = .false.

  POST has output on full grids.
- logical, public num_vars::post_output_sol = .false.

  POST has outputs of solution.
- logical, public num_vars::do_execute_command_line = .false.

  call "execute_command_line" inside program
- logical, public num_vars::print_mem_usage = .false.

  print memory usage is printed
- logical, public num_vars::swap_angles = .false.

  swap angles theta and zeta in plots (only for POST)
- logical, public num_vars::compare_tor_pos = .false.

  compare quantities at toroidal positions (only for POST)
- logical, public num_vars::retain_all_sol

  retain also faulty solutions
- character(len=5), public num_vars::plot_dir = 'Plots'

  directory where to save plots
- character(len=7), public num_vars::script_dir = 'Scripts'

  directory where to save scripts for plots
- character(len=4), public num_vars::data_dir = 'Data'

  directory where to save data for plots
- integer, public num_vars::n_theta_plot

  nr. of poloidal points in plot
- integer, public num_vars::n_zeta_plot

  nr. of toroidal points in plot
- integer, dimension(2), public num_vars::n_vac_plot

  nr. of points in R and Z in vacuum
- real(dp), public num_vars::min_theta_plot

  min. of `theta_plot`
- real(dp), public num_vars::max_theta_plot

  max. of `theta_plot`
- real(dp), public num_vars::min_zeta_plot

  min. of `zeta_plot`
- real(dp), public num_vars::max_zeta_plot

  max. of `zeta_plot`
- real(dp), public num_vars::min_r_plot

  min. of `r_plot`
- real(dp), public num_vars::max_r_plot

  max. of `r_plot`
- real(dp), public num_vars::min_rvac_plot

  min. of `R` in which to plot vacuum

- real(dp), public num_vars::max_rvac_plot

  max. of R in which to plot vacuum
- real(dp), public num_vars::min_zvac_plot

  min. of Z in which to plot vacuum
- real(dp), public num_vars::max_zvac_plot

  max. of Z in which to plot vacuum
- integer, public num_vars::plot_grid_style

  style for POST plot grid (0: 3-D plots, 1: slab plots, 2: slab plots with folding, 3: straight cylinder))
- integer, public num_vars::n_sol_requested

  how many solutions requested
- integer, dimension(4), public num_vars::n_sol_plotted

  how many solutions to be plot (first unstable, last unstable, first stable, last stable)
- integer, dimension(2), public num_vars::plot_size

  size of plot in inches
- integer, public num_vars::rich_restart_lvl

  starting Richardson level (0: none [default])
- character(len=max_str_ln), public num_vars::input_name

  will hold the full name of the input file
- integer, public num_vars::ex_plot_style

  external plot style (1: GNUPlot, 2: Bokeh for 2D, Mayavi for 3D)
- real(dp), public num_vars::pert_mult_factor_post

  factor with which to multiply perturbation strength for POST
- real(dp), dimension(2), public num_vars::rz_0

  origin of geometrical poloidal coordinate
- integer, parameter, public num_vars::input_i = 50

  file number of input file
- integer, parameter, public num_vars::eq_i = 51

  file number of equilibrium file from VMEC or HELENA
- integer, parameter, public num_vars::pb3d_i = 52

  file number of PB3D output file
- integer, parameter, public num_vars::output_i = 53

  file number of output file
- integer, parameter, public num_vars::shell_commands_i = 54

  file number of shell commands file
- integer, parameter, public num_vars::mem_usage_i = 55

  file number of memory usage file
- integer, parameter, public num_vars::output_ev_i = 56

  file number of output of EV
- integer, parameter, public num_vars::decomp_i = 57

  file number of output of EV decomposition
- integer, parameter, public num_vars::hel_export_i = 59

  file number of output of HELENA equilibrium export file
- integer, parameter, public num_vars::hel_pert_i = 58

  file number of HELENA equilibrium perturbation file
- integer, parameter, public num_vars::prop_b_tor_i = 60

  file number of B_ϕ proportionality factor file

D.61 num_vars.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Numerical variables used by most other modules.
!-----------------------------------------------------------------------------!
module num_vars
    use iso_fortran_env

    implicit none
    private
    public dp, dpi, max_str_ln, max_name_ln, max_deriv, prog_name, &
        &output_name, prog_version, prog_style, min_pb3d_version, &
        &debug_version, &
        &shell_commands_name, mem_usage_name, &
        &mem_usage_count, weight_dp, rank, n_procs, sol_n_procs, time_start, &
        &max_tot_mem, max_x_mem, x_jobs_lims, x_job_nr, &
        &eq_jobs_lims, eq_job_nr, mem_scale_fac, pi, mu_0_original, iu, &
        &ev_style, eq_style, rho_style, u_style, norm_style, bc_style, &
        &x_style, x_grid_style, matrix_slepc_style, plot_resonance, &
        &plot_magn_grid, plot_b, plot_j, plot_flux_q, plot_kappa, plot_sol_xi, &
        &plot_sol_q, plot_e_rec, plot_vac_pot, ltest, use_pol_flux_e, &
        &use_pol_flux_f, use_normalization, ev_bc, tol_slepc, max_it_slepc, &
        &norm_disc_prec_eq, k_style, ev_guess, norm_disc_prec_x, &
        &norm_disc_prec_sol, norm_disc_style_sol, &
        &post_style, alpha_style, magn_int_style, solver_slepc_style, &
        &max_njq_change, &
        &max_it_rich, tol_rich, max_it_zero, max_nr_backtracks_hh, &
        &tol_zero, tol_norm, &
        &ex_max_size, eq_name, &
        &no_plots, no_output, plot_dir, script_dir, data_dir, n_theta_plot, &
        &n_zeta_plot, min_theta_plot, max_theta_plot, min_zeta_plot, &
        &max_zeta_plot, min_r_plot, max_r_plot, n_sol_requested, &
        &n_sol_plotted, retain_all_sol, do_execute_command_line, &
        &print_mem_usage, input_name, plot_grid_style, swap_angles, &
        &rich_restart_lvl, plot_size, jump_to_sol, compare_tor_pos, &
        &export_hel, plot_vmec_modes, ex_plot_style, pert_mult_factor_post, &
        &min_rvac_plot, max_rvac_plot, min_zvac_plot, max_zvac_plot, &
        &n_vac_plot, invert_top_bottom_h, &
        &rz_0, &
        &post_output_full, post_output_sol, &
        &shell_commands_i, mem_usage_i, output_ev_i, decomp_i, &
        &hel_pert_i, hel_export_i, input_i, pb3d_i, pb3d_name, eq_i, output_i, &
        &prop_b_tor_i

    ! technical variables
    !integer, parameter :: dp = kind(1.d0)                                   !< double precision
    !integer, parameter :: qp = selected_real_kind (32)                      !< quadruple precision
    integer, parameter :: dp = real64                                       !< double precision
    !integer, parameter :: qp = REAL128                                      !< quadruple precision
    integer, parameter :: dpi = int64                                       !< double precision
    real(dp), parameter :: weight_dp = 0.008                                !< size of double precision in kB
    integer, parameter :: max_str_ln = 120                                  !< maximum length of strings
    integer, parameter :: max_name_ln = 30                                  !< maximum length of filenames
    integer, parameter :: max_deriv = 2                                     !< highest derivatives for metric factors in Flux coords.
    integer :: prog_style                                                   !< program style (1: PB3D, 2: PB3D_POST)
    character(len=4) :: prog_name                                           !< name of program, used for info
    character(len=3), parameter :: output_name = 'out'                      !< name of output file
    character(len=14), parameter :: shell_commands_name = 'shell_commands'  !< name of shell commands file
    character(len=9), parameter :: mem_usage_name = 'mem_usage'             !< name of memory usage file
    integer :: mem_usage_count                                              !< counter for memory usage output
    real(dp), parameter :: prog_version = 2.47_dp                           !< version number
    real(dp), parameter :: min_pb3d_version = 2.43_dp                       !< minimum PB3D version for POST
#if ldebug
    logical :: debug_version = .true.                                       !< debug version used
#else
    logical :: debug_version = .false.                                      !< not debug version
#endif

    ! MPI variables
    integer :: rank                                                         !< MPI rank
    integer :: n_procs                                                      !< nr. of MPI processes
    integer :: sol_n_procs                                                  !< nr. of MPI processes for solution with SLEPC
    integer(kind=8) :: time_start                                           !< start time of simulation

    ! job variables
    real(dp) :: max_tot_mem                                                 !< maximum total memory for all processes [MB]
    real(dp) :: max_x_mem                                                   !< maximum memory for perturbation calculations for all processes [MB]
    integer, allocatable :: x_jobs_lims(:,:)                                !< data about X jobs: [\f$\min_k\f$, \f$\max_k\f$, \f$\min_m\f$, \f$\max_m\f$] for all jobs
```

```
    integer, allocatable :: eq_jobs_lims(:,:)                    !< data about eq jobs: [\f$\min_\theta\f$, \f$\max_\theta\f$] for all jobs
    integer :: x_job_nr                                          !< nr. of X job
    integer :: eq_job_nr                                         !< nr. of eq job
    real(dp), parameter :: mem_scale_fac = 6.0                   !< empirical scale factor of memory to calculate eq compared to just storing it

    ! physical and mathematical variables
    real(dp), parameter :: pi=4_dp*datan(1.0_dp)                 !< \f$\pi\f$
    real(dp), parameter :: mu_0_original = 4e-7_dp*pi            !< permeability of free space
    complex(dp), parameter :: iu = (0,1)                         !< complex unit

    ! concerning runtime
    integer :: ev_style                                          !< determines the method used for solving an EV problem
    integer :: eq_style                                          !< either 1 (VMEC) or 2 (HELENA)
    integer :: rho_style                                         !< style for equilibrium density profile
    integer :: u_style                                           !< style for calculation of U (1: ord.2, 2: ord.1, 1: ord.0)
    integer :: norm_style                                        !< style for normalization
    integer :: k_style                                           !< style for kinetic energy
    integer :: bc_style(2)                                       !< style for BC left and right
    integer :: x_style                                           !< style for secondary mode numbers (1: prescribed, 2: fast)
    integer :: matrix_slepc_style                                !< style for matrix storage (1: sparse, 2: shell)
    integer :: solver_slepc_style                                !< style for solver (1: Krylov-Schur, 2: GD)
    integer :: post_style                                        !< style for POST (1: extended grid, 2: B-aligned grid)
    integer :: x_grid_style                                      !< style for normal component of X grid (1: eq, 2: sol, 3: enriched)
    integer :: alpha_style                                       !< style for alpha (1: one field line, many turns, 2: many field lines, one turn)
    integer :: max_it_slepc                                      !< maximum nr. of iterations for SLEPC
    logical :: plot_resonance                                    !< whether to plot the q-profile or iota-profile with resonances
    logical :: plot_magn_grid                                    !< whether to plot the grid in real coordinates
    logical :: plot_b                                            !< whether to plot the magnetic field in real coordinates
    logical :: plot_j                                            !< whether to plot the current in real coordinates
    logical :: plot_flux_q                                       !< whether to plot flux quantities in real coordinates
    logical :: plot_kappa                                        !< whether to plot curvature
    logical :: plot_sol_xi                                       !< whether to plot plasma perturbation of solution in POST
    logical :: plot_sol_q                                        !< whether to plot magnetic perturbation of solution in POST
    logical :: plot_vac_pot                                      !< whether to plot vacuum potential in POST
    logical :: plot_e_rec                                        !< whether to plot energy reconstruction in POST
    logical :: ltest                                             !< whether or not to call the testing routines
    logical :: use_pol_flux_e                                    !< whether poloidal flux is used in E coords.
    logical :: use_pol_flux_f                                    !< whether poloidal flux is used in F coords.
    logical :: use_normalization                                 !< whether to use normalization or not
    real(dp) :: ev_bc                                            !< value of artificial Eigenvalue for boundary condition
    real(dp) :: ev_guess                                         !< first guess for eigenvalue
    real(dp), allocatable :: tol_slepc(:)                        !< tolerance for SLEPC for different Richardson levels
    real(dp) :: max_njq_change                                   !< maximum change of prim. mode number times saf.  fac. / rot. transf. when using X_style 2 (fast)
    integer :: norm_disc_prec_eq                                 !< precision for normal discretization for equilibrium
    integer :: norm_disc_prec_x                                  !< precision for normal discretization for perturbation
    integer :: norm_disc_prec_sol                                !< precision for normal discretization for solution
    integer :: norm_disc_style_sol                               !< style for normal discretization for solution (1: central fin. diff., 2: left fin. diff.)
    integer :: magn_int_style                                    !< style for magnetic integrals (1: trapezoidal, 2: Simpson 3/8)

    ! concerning Richardson extrapolation
    integer :: max_it_rich                                       !< number of levels for Richardson extrapolation
    real(dp) :: tol_rich                                         !< tolerance for Richardson extrapolation

    ! concerning finding the magnetic field lines
    integer :: max_it_zero                                       !< maximum number of iterations to find zeros
    integer :: max_nr_backtracks_hh                              !< maximum number of backtracks for Householder, relax. factors
```

```
    real(dp) :: tol_zero                                        !< tolerance for zeros
    real(dp) :: tol_norm                                        !< tolerance for normal range (normalized to 0..1)

    ! concerning input / output
    integer, parameter :: ex_max_size = 10000                   !< maximum size of matrices for external plot
    character(len=max_str_ln) :: eq_name                        !< name of equilibrium file from VMEC or HELENA
    character(len=max_str_ln) :: pb3d_name                      !< name of PB3D output file
    logical :: no_plots = .false.                               !< no plots made
    logical :: jump_to_sol = .false.                            !< jump to solution
    logical :: export_hel = .false.                             !< export HELENA
    logical :: plot_vmec_modes = .false.                        !< plot VMEC modes
    logical :: invert_top_bottom_h = .false.                    !< invert top and bottom for HELENA equilibria
    logical :: no_output = .false.                              !< no output shown
    logical :: post_output_full = .false.                       !< POST has output on full grids
    logical :: post_output_sol = .false.                        !< POST has outputs of solution
    logical :: do_execute_command_line = .false.                !< call "execute_command_line" inside program
    logical :: print_mem_usage = .false.                        !< print memory usage is printed
    logical :: swap_angles = .false.                            !< swap angles theta and zeta in plots (only for POST)
    logical :: compare_tor_pos = .false.                        !< compare quantities at toroidal positions (only for POST)
    logical :: retain_all_sol                                   !< retain also faulty solutions
    character(len=5) :: plot_dir = 'Plots'                      !< directory where to save plots
    character(len=7) :: script_dir = 'Scripts'                  !< directory where to save scripts for plots
    character(len=4) :: data_dir = 'Data'                       !< directory where to save data for plots
    integer :: n_theta_plot                                     !< nr. of poloidal points in plot
    integer :: n_zeta_plot                                      !< nr. of toroidal points in plot
    integer :: n_vac_plot(2)                                    !< nr. of points  in R and Z in vacuum
    real(dp) :: min_theta_plot                                  !< min. of \c theta_plot
    real(dp) :: max_theta_plot                                  !< max. of \c theta_plot
    real(dp) :: min_zeta_plot                                   !< min. of \c zeta_plot
    real(dp) :: max_zeta_plot                                   !< max. of \c zeta_plot
    real(dp) :: min_r_plot                                      !< min. of \c r_plot
    real(dp) :: max_r_plot                                      !< max. of \c r_plot
    real(dp) :: min_rvac_plot                                   !< min. of \c R in which to plot vacuum
    real(dp) :: max_rvac_plot                                   !< max. of \c R in which to plot vacuum
    real(dp) :: min_zvac_plot                                   !< min. of \c Z in which to plot vacuum
    real(dp) :: max_zvac_plot                                   !< max. of \c Z in which to plot vacuum
    integer :: plot_grid_style                                  !< style for POST plot grid (0: 3-D plots, 1: slab plots, 2: slab plots with folding, 3: straight cylinder))
    integer :: n_sol_requested                                  !< how many solutions requested
    integer :: n_sol_plotted(4)                                 !< how many solutions to be plot (first unstable, last unstable, first stable, last stable)
    integer :: plot_size(2)                                     !< size of plot in inches
    integer :: rich_restart_lvl                                 !< starting Richardson level (0: none [default])
    character(len=max_str_ln) :: input_name                     !< will hold the full name of the input file
    integer :: ex_plot_style                                    !< external plot style (1: GNUPlot, 2: Bokeh for 2D, Mayavi for 3D)
    real(dp) :: pert_mult_factor_post                           !< factor with which to multiply perturbation strength for POST

    ! Concerning comparing toroidal positions
    real(dp) :: rz_0(2)                                         !< origin of geometrical poloidal coordinate

    ! Concerning file numbers
    integer, parameter :: input_i = 50                          !< file number of input file
    integer, parameter :: eq_i = 51                             !< file number of equilibrium file from VMEC or HELENA
    integer, parameter :: pb3d_i = 52                           !< file number of PB3D output file
    integer, parameter :: output_i = 53                         !< file number of output file
    integer, parameter :: shell_commands_i = 54                 !< file number of shell commands file
    integer, parameter :: mem_usage_i = 55                      !< file number of memory usage file
    integer, parameter :: output_ev_i = 56                      !< file number of output of EV
    integer, parameter :: decomp_i = 57                         !< file number of output of EV decomposition
    integer, parameter :: hel_export_i = 59                     !< file number of output of HELENA equilibrium export file
    integer, parameter :: hel_pert_i = 58                       !< file number of HELENA equilibrium perturbation file
    integer, parameter :: prop_b_tor_i = 60                     !< file number of \f$B_\phi\f$ proportionality factor file
end module num_vars
```

D.62 /Users/toonweyens/Code/PB3D/Modules/output_ops.f90 File Reference

Interfaces and Types

- interface output_ops::print_ex_2d

  Print 2-D output on a file.
- interface output_ops::print_ex_3d

  Print 3-D output on a file.
- interface output_ops::plot_hdf5

  Prints variables `vars` with names `var_names` in an HDF5 file with name c file_name and accompanying XDMF file.

Modules

- module output_ops

  Operations concerning giving output, on the screen as well as in output files.

Functions/Subroutines

- subroutine, public output_ops::draw_ex (var_names, draw_name, nplt, draw_dim, plot_on_screen, ex↩_plot_style, data_name, draw_ops, extra_ops, is_animated, ranges, delay, persistent)

  Use external program to draw a plot.
- subroutine, public output_ops::plot_diff_hdf5 (a, b, file_name, tot_dim, loc_offset, descr, output_↩message)

  Takes two input vectors and plots these as well as the relative and absolute difference in a HDF5 file.

D.63 output_ops.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------!
!> Operations  concerning giving  output, on  the screen  as well  as in  output
!! files.
!-----------------------------------------------------------------------!
module output_ops
#include <PB3D_macros.h>
    use str_utilities
    use messages
    use num_vars, only: dp, max_str_ln, no_plots, iu, plot_dir, data_dir, &
        &script_dir, plot_size
    use files_utilities, only: nextunit
    use grid_vars, only: grid_type


    implicit none
    private
    public print_ex_2d, print_ex_3d, draw_ex, plot_hdf5, plot_diff_hdf5

    ! global variables
    integer :: temp_id(2) = 1                                          !< \private will be appended to temporary data and script files
    character(len=9) :: line_clrs(20) = [&
        &'"#7297E6"','"#67EB84"','"#F97A6D"','"#F9C96D"','"#1D4599"',&
        &'"#11AD34"','"#E62B17"','"#E69F17"','"#2F3F60"','"#2F6C3D"',&
        &'"#8F463F"','"#8F743F"','"#031A49"','"#025214"','"#6D0D03"',&
        &'"#6D4903"','"#A9BDE6"','"#A6EBB5"','"#F9B7B0"','"#F9E0B0"']   !< \private line colors, from <https://github.com/Gnuplotting/gnuplot-palettes>
    character(len=max_str_ln) :: line_style = 'lt 1 lw 1 pt 7 ps 0.5;'  !< \private line style without little space in line connecting points
```

```
#if ldebug
    character(len=0) :: err_output_str = ''                                  !< \private string with error output
#else
    character(len=14) :: err_output_str = ' 2> /dev/null'                    !< \private string with error output (/dev/null)
#endif

    ! interfaces

    !> \public Print 2-D output on a file.
    !!
    !! The variables \c var_name and \c file_name  hold the name of the plot and
    !! of the file in which the plot data  is to be saved, respectively. \c y is
    !! the an  array containing  the function  which is  stored and  \c x  is an
    !! optional vector  with the  x-values. The logical  \c draw  can optionally
    !! disable  calling the  external  drawing procedure  for  output on  screen
    !! [default], without modifying the plot file.
    !!
    !! The first index of \c y (and \c x) contains the points of a current plot.
    !!
    !! The second index indicates various plots (one or more)
    interface print_ex_2d
        !> \public
        module procedure print_ex_2d_ind
        !> \public
        module procedure print_ex_2d_arr
    end interface

    !> \public Print 3-D output on a file.
    !!
    !! The variables \c var_name and \c file_name  hold the name of the plot and
    !! of the file in which the plot data  is to be saved, respectively. \c z is
    !! the an array containing the function which is stored and \c x and\c y are
    !! optional  vectors with  the  x  and y-values.  The  logical  \c draw  can
    !! optionally disable calling  the external drawing procedure  for output on
    !! screen [default], without modifying the plot file.
    !!
    !! The first index of \c z (and \c x, \c y) contains the points of a current
    !! The plot second index indicates various plots (one or more)
    interface print_ex_3d
        !> \public
        module procedure print_ex_3d_ind
        !> \public
        module procedure print_ex_3d_arr
    end interface


    !> \public Prints variables \c vars with  names \c var_names in an HDF5 file
    !! with name c file_name and accompanying XDMF file.
    !!
    !! For XDMF collections (\cite xdmf), only  the first value for \c var_names
    !! is used, so it should have a size of one.
    !!
    !! The plot is generally 3-D, but if one of the dimensions provided is equal
    !! to 1, it is checked whether there is poloidal or toroidal axisymmetry and
    !! if so, the plot becomes 2-D. This  can be forced using the optional input
    !! argument \c sym_type.
    !!
    !! Optionally, the (curvilinear) grid can be provided through \c X, \c Y and
    !! \c Z. If not,  the grid is assumed to be  Cartesian with discrete indices
    !! where \c X corresponds to the first dimensions, \c Y to the second and \c
    !! Z to the third.
    !!
    !! Additionally, the total grid size and  local offset can be provided in \c
    !! tot_dim and \c loc_offset to  run this routine in parallel automatically.
    !! Optionally,  one  of  the  dimensions  (\c  col_id,  default  4)  can  be
    !! associated to a collection dimension using \c col different from 1:
    !!  - \c col = 1: no collection, just plots of different variables
    !!  - \c col = 2: time collection
    !!  - \c col = 3: spatial collection
    !!  - \c col = 4: vector field
    !!
    !! Furthermore,  using  the  variable  \c  cont_plot,  a  plot  can  be
    !! (over-)written in multiple writes. By this  is meant that there should be
    !! an initial  plot, with collection type  1, 2, 3  or 4, which can  then be
    !! followed by an arbitrary number of additional writes. As these additional
    !! writes  currently  cannot  modify  the  plot  structure,  nor  the  XDMF
    !! variables, their collection dimension should be complete from the start.
    !!
    !! This has no  implications for single plots but means  that for collection
    !! types 2  to 4  all the  elements in  the collection  have to  be present,
    !! though they do not necessary need  to have been completely written in the
    !! other dimensions.
    !!
    !! Subsequent writes with  \c cont_plot can then, for  instance, write parts
    !! of the data  that had not yet  been written, or overwrite  ones that had.
    !! This can be useful for  post-processing where the memory requirements are
```

```
    !! large so that the work has to be split.
    !!
    !! \note
    !!  -# For a vector field, the number of variables has to be 2 for 2-D plots
    !!  or 3  for 3-D  plots. This should  be rewritten in  the future,  so that
    !!  collections can be used for vectors as well.
    !!  -# In order to merge collections in their collection dimension, the XDMF
    !!  files can always easily be joined.
    !!  -# If  necessary, a lock system  should be used when  multiple processes
    !!  are writing the same file, including continued writes.
    !!  -# To plot this with VisIt, use:
    !!      -  for temporal  collections:  pseudocolor using  the variable  name
    !!      (other names are ignored).
    !!      - for spatial collections: subset of blocks or pseudocolor using the
    !!      variable name (other names are ignored).
    !!      - for vector plot: Vector plot.
    !!      - for without collections: pseudocolor using the variable names.
    !!  -# Currently all of possibly multiple  processes that write data have to
    !!  cover the entire range of the variables in the dimension indicated by \c
    !!  col_id. (This could be implemented by changing how n_plot is defined and
    !!  selectively  letting  each processer  write  in  the  main loop  at  its
    !!  corresponding indices.)
    !!  -# To project  the data to 2-D  in VisIt, use the  projection tool under
    !!  Operators > Transform
    interface plot_hdf5
        !> \public
        module procedure plot_hdf5_ind
        !> \public
        module procedure plot_hdf5_arr
    end interface

contains
    !> \private individual version
    subroutine print_ex_2d_ind(var_name,file_name_i,y,x,draw,persistent)
        ! input / output
        character(len=*), intent(in) :: var_name                                !< name of variable in legend
        character(len=*), intent(in) :: file_name_i                             !< name of input file
        real(dp), intent(in) :: y(1:)                                           !< ordinate
        real(dp), intent(in), optional :: x(1:)                                 !< absicca
        logical, intent(in), optional :: draw                                   !< whether to draw the plot as well
        logical, intent(in), optional :: persistent                             !< keep on-screen plot open

        ! local variables
        integer :: npoints

        ! set npoints, ny
        npoints = size(y)

        ! call multiplot version
        if (present(x)) then
            call print_ex_2d_arr([var_name],file_name_i,reshape(y,[npoints,1]),&
                &x=reshape(x,[npoints,1]),draw=draw,persistent=persistent)
        else
            call print_ex_2d_arr([var_name],file_name_i,reshape(y,[npoints,1]),&
                &draw=draw,persistent=persistent)
        end if
    end subroutine print_ex_2d_ind
    !> \private array version
    subroutine print_ex_2d_arr(var_names,file_name_i,y,x,draw,persistent)
        use num_vars, only: rank

        ! input / output
        character(len=*), intent(in) :: var_names(:)                            !< names of variables in legend
        character(len=*), intent(in) :: file_name_i                             !< name of input file
        real(dp), intent(in) :: y(1:,1:)                                        !< ordinate
        real(dp), intent(in), optional :: x(1:,1:)                              !< absicca
        logical, intent(in), optional :: draw                                   !< whether to draw the plot as well
        logical, intent(in), optional :: persistent                             !< keep on-screen plot open

        ! local variables
        integer :: file_i
        integer :: iplt, ipnt, nplt, npnt
        real(dp), allocatable :: x_fin(:,:)
        integer :: istat
        character(len=max_str_ln) :: file_name
        character(len=max_str_ln) :: write_format
        logical :: persistent_loc                                               ! local persistent

        ! set nplt, npnt
        npnt = size(y,1)
        nplt = size(y,2)
        if (present(x)) then
            if (size(x,1).ne.size(y,1) .or. &
                &(size(x,2).ne.1 .and. size(x,2).ne.size(y,2))) then
                write(*,*) 'shape(x)', shape(x)
                write(*,*) 'shape(y)', shape(y)
                call writo('In print_ex_2D, the size of x and y has to be the &
```

```
                    &same... Skipping plot',persistent=.true.,warning=.true.)
            end if
        end if

        ! set other variables
        persistent_loc = .false.
        if (present(persistent)) persistent_loc = persistent

        ! set x
        allocate(x_fin(npnt,nplt))
        if (present(x)) then
            if (size(x,2).eq.1) then                                        ! assume copy
                do iplt = 1,nplt
                    x_fin(:,iplt) = x(:,1)
                end do
            else
                x_fin = x
            end if
        else
            do ipnt = 1,npnt
                x_fin(ipnt,:) = ipnt
            end do
        end if

        ! set default file name if empty
        if (trim(file_name_i).eq.'') then
            file_name = 'temp_data_print_ex_2D_'//trim(i2str(rank))//'_'//&
                trim(i2str(temp_id(1)))
            temp_id(1) = temp_id(1)+1
        else
            file_name = trim(file_name_i)
        end if

        ! open output file
        open(unit=nextunit(file_i),file=data_dir//'/'//trim(file_name)//'.dat',&
            &iostat=istat)
        chckstt

        ! write to output file
        write(file_i,'(1X,A)',iostat=istat) &
            &'# '//trim(merge_strings(var_names))//':'
        write_format="("//trim(i2str(nplt*2))//"(ES23.16,' '))"
        do ipnt = 1,npnt
            write(file_i,fmt=trim(write_format),iostat=istat) &
                &(x_fin(ipnt,iplt), iplt = 1,nplt), &
                &(y(ipnt,iplt), iplt = 1,nplt)
        enddo
        write(file_i,'(1X,A)',iostat=istat) ''

        ! close output file
        close(file_i)

        ! bypass plots if no_plots
        if (no_plots) return

        ! if draw is present and equal to .false., cancel calling draw_ex
        if (present(draw)) then
            if (.not.draw) return
        end if
        call draw_ex(var_names,file_name,nplt,1,.true.,persistent=persistent)

        if (trim(file_name_i).eq.'' .and. .not.persistent_loc) then
            call use_execute_command_line('rm '//data_dir//'/'//&
                &trim(file_name)//'.dat')
        end if
    end subroutine print_ex_2d_arr

    !> \private individual version
    subroutine print_ex_3d_ind(var_name,file_name_i,z,y,x,draw)
        ! input / output
        character(len=*), intent(in) :: var_name                            !< name of variable in legend
        character(len=*), intent(in) :: file_name_i                         !< name of input file
        real(dp), intent(in) :: z(1:,1:)                                    !< ordinate
        real(dp), intent(in), optional :: x(1:,1:)                          !< absicca
        real(dp), intent(in), optional :: y(1:,1:)                          !< absicca
        logical, intent(in), optional :: draw                               !< whether to draw the plot as well

        ! local variables
        integer :: npoints(2)

        ! set npoints, ny
        npoints = [size(z,1),size(z,2)]

        ! call multiplot version
        if (present(y)) then
            if (present(x)) then
                call print_ex_3d_arr([var_name],file_name_i,&
```

```
                    &reshape(z,[size(z,1),size(z,2),1]),&
                    &y=reshape(y,[size(z,1),size(z,2),1]),&
                    &x=reshape(x,[size(z,1),size(z,2),1]),draw=draw)
            else
                call print_ex_3d_arr([var_name],file_name_i,&
                    &reshape(z,[size(z,1),size(z,2),1]),&
                    &y=reshape(y,[size(z,1),size(z,2),1]),&
                    &draw=draw)
            end if
        else
            if (present(x)) then
                call print_ex_3d_arr([var_name],file_name_i,&
                    &reshape(z,[size(z,1),size(z,2),1]),&
                    &x=reshape(x,[size(z,1),size(z,2),1]),&
                    &draw=draw)
            else
                call print_ex_3d_arr([var_name],file_name_i,&
                    &reshape(z,[size(z,1),size(z,2),1]),draw=draw)
            end if
        end if
    end subroutine print_ex_3d_ind
    !> \private array version
    subroutine print_ex_3d_arr(var_names,file_name_i,z,x,y,draw)
        use num_vars, only: rank

        ! input / output
        character(len=*), intent(in) :: var_names(:)                          !< names of variables in legend
        character(len=*), intent(in) :: file_name_i                           !< name of input file
        real(dp), intent(in) :: z(1:,1:,1:)                                   !< ordinate
        real(dp), intent(in), optional :: x(1:,1:,1:)                         !< absicca
        real(dp), intent(in), optional :: y(1:,1:,1:)                         !< absicca
        logical, intent(in), optional :: draw                                 !< whether to draw the plot as well

        ! local variables
        integer :: file_i
        integer :: iplt, ipntx, ipnty, nplt, npntx, npnty
        real(dp), allocatable :: x_fin(:,:,:)
        real(dp), allocatable :: y_fin(:,:,:)
        integer :: istat
        character(len=max_str_ln) :: file_name
        character(len=max_str_ln) :: write_format

        ! set nplt, npnt
        npntx = size(z,1)
        npnty = size(z,2)
        nplt = size(z,3)
        if (present(x)) then
            if (size(x,1).ne.size(z,1) .or. size(x,2).ne.size(z,2) .or. &
                &(size(x,3).ne.1 .and. size(x,3).ne.size(z,3))) then
                write(*,*) 'shape(x)', shape(x)
                write(*,*) 'shape(z)', shape(z)
                call writo('In print_ex, the size of x and z has to be the &
                    &same... Skipping plot',persistent=.true.,warning=.true.)
                return
            end if
        end if
        if (present(y)) then
            if (size(y,1).ne.size(z,1) .or. size(y,2).ne.size(z,2) .or. &
                &(size(y,3).ne.1 .and. size(y,3).ne.size(z,3))) then
                write(*,*) 'shape(y)', shape(y)
                write(*,*) 'shape(z)', shape(z)
                call writo('In print_ex, the size of y and z has to be the &
                    &same... Skipping plot',persistent=.true.,warning=.true.)
                return
            end if
        end if

        ! set x
        allocate(x_fin(npntx,npnty,nplt))
        if (present (x)) then
            if (size(x,3).eq.1) then                                          ! assume copy
                do iplt = 1,nplt
                    x_fin(:,:,iplt) = x(:,:,1)
                end do
            else
                x_fin = x
            end if
        else
            do ipntx = 1,npntx
                x_fin(ipntx,:,:) = ipntx
            end do
        end if
        ! set y
        allocate(y_fin(npntx,npnty,nplt))
        if (present (y)) then
            if (size(y,3).eq.1) then                                          ! assume copy
                do iplt = 1,nplt
```

```
                    y_fin(:,:,iplt) = y(:,:,1)
                end do
            else
                y_fin = y
            end if
        else
            do ipnty = 1,npnty
                y_fin(:,ipnty,:) = ipnty
            end do
        end if

        ! set default file name if empty
        if (trim(file_name_i).eq.'') then
            file_name = 'temp_data_print_ex_3D_'//trim(i2str(rank))//'_'//&
                trim(i2str(temp_id(1)))
            temp_id(1) = temp_id(1)+1
        else
            file_name = trim(file_name_i)
        end if

        ! open output file
        open(nextunit(file_i),file=data_dir//'/'//trim(file_name)//'.dat',&
            &iostat=istat)
        chckstt

        ! write to output file
        write(file_i,'(A)',iostat=istat) &
            &'# '//trim(merge_strings(var_names))//':'
        write_format="("//trim(i2str(nplt*3))//"(ES23.16,' '))"
        do ipntx = 1,npntx
            do ipnty = 1,npnty
                write(file_i,fmt=trim(write_format),iostat=istat) &
                    &(x_fin(ipntx,ipnty,iplt), iplt = 1,nplt), &
                    &(y_fin(ipntx,ipnty,iplt), iplt = 1,nplt), &
                    &(z(ipntx,ipnty,iplt), iplt = 1,nplt)
            end do
            write(file_i,'(A)',iostat=istat) ''
        enddo
        write(file_i,'(A)',iostat=istat) ''

        ! close output file
        close(file_i)

        ! bypass plots if no_plots
        if (no_plots) return

        ! if draw is present and equal to .false., cancel calling draw_ex
        if (present(draw)) then
            if (.not.draw) return
        end if
        call draw_ex(var_names,file_name,nplt,2,.true.)

        if (trim(file_name_i).eq.'') then
            call use_execute_command_line('rm '//data_dir//'/'//&
                &trim(file_name)//'.dat')
        end if
    end subroutine print_ex_3d_arr

    !> \private array version
    subroutine plot_hdf5_arr(var_names,file_name,vars,tot_dim,loc_offset,&
        &X,Y,Z,col_id,col,sym_type,cont_plot,descr)
        use hdf5_ops, only: open_hdf5_file, add_hdf5_item, print_hdf5_top, &
            &print_hdf5_geom, print_hdf5_3d_data_item, print_hdf5_att, &
            &print_hdf5_grid, close_hdf5_file, merge_hdf5_3d_data_items
        use hdf5_vars, only: dealloc_xml_str, &
            &xml_str_type, hdf5_file_type
        use num_vars, only: n_procs
        use mpi_utilities, only: get_ser_var

        ! input / output
        character(len=*), intent(in) :: var_names(:)                    !< names of variable to be plot
        character(len=*), intent(in) :: file_name                       !< file name
        real(dp), intent(in), target :: vars(:,:,:,:)                   !< variables to plot
        integer, intent(in), optional :: tot_dim(4)                     !< total dimensions of the arrays
        integer, intent(in), optional :: loc_offset(4)                  !< offset of local dimensions
        real(dp), intent(in), target, optional :: X(:,:,:,:)            !< curvilinear grid X points
        real(dp), intent(in), target, optional :: Y(:,:,:,:)            !< curvilinear grid Y points
        real(dp), intent(in), target, optional :: Z(:,:,:,:)            !< curvilinear grid Z points
        integer, intent(in), optional :: col_id                         !< index of time dimension
        integer, intent(in), optional :: col                            !< whether a collection is made
        integer, intent(in), optional :: sym_type                       !< type of symmetry (1: no symmetry, 2:
    toroidal, 3: poloidal)
        logical, intent(in), optional :: cont_plot                      !< continued plot
        character(len=*), intent(in), optional :: descr                 !< description

        ! local variables
        type(hdf5_file_type) :: file_info                               ! file info
```

```
        integer :: istat                                                ! status
        integer :: col_id_loc                                           ! local copy of col_id
        integer :: col_loc                                              ! local copy of col
        integer :: n_plot                                               ! nr. of plots
        integer :: id, jd                                               ! counter
        integer :: sym_type_loc                                         ! local sym_type
        integer :: sym_pol, sym_tor                                     ! used to determine symmetry
        integer :: att_id                                               ! index in att
        integer, allocatable :: tot_sym_pol(:), tot_sym_tor(:)          ! sym_pol and sym_tor for all processes
        integer :: tot_dim_loc(4)                                       ! local copy of tot_dim
        integer :: loc_offset_loc(4)                                    ! local copy of loc_offset
        integer :: tot_dim_3D(3)                                        ! tot_dim except collection
        integer :: loc_dim_3D(3)                                        ! loc_dim except collection
        integer :: loc_offset_3D(3)                                     ! loc_offset except collection
        type(xml_str_type) :: col_grid                                  ! grid with collection
        type(xml_str_type), allocatable :: grids(:)                     ! the grids in the time collection
        type(xml_str_type) :: top                                       ! topology
        type(xml_str_type), allocatable :: dat(:)                       ! data items for geometry and attribute
        type(xml_str_type) :: dat_vec                                   ! data items for merged attribute
        type(xml_str_type) :: geom                                      ! geometry
        type(xml_str_type) :: att(1)                                    ! attribute
        logical :: col_mask(4)                                          ! to select out the collection dimension
        logical :: ind_plot                                             ! individual plot
        logical :: cont_plot_loc                                        ! local cont_plot
        logical :: first_vc, last_vc                                    ! first or last vector components (for the
    others, some XDMF operations can be skipped)
        real(dp), allocatable :: sym_ang(:,:,:)                         ! angle to be checked for symmetry
        real(dp) :: tol_sym = 1.e-8_dp                                  ! tolerance for symmetry determination
        real(dp), pointer :: var_3D(:,:,:) => null()                    ! pointer to vars
        real(dp), pointer :: X_3D(:,:,:) => null()                      ! pointer to X
        real(dp), pointer :: Y_3D(:,:,:) => null()                      ! pointer to Y
        real(dp), pointer :: Z_3D(:,:,:) => null()                      ! pointer to Z
        character(len=max_str_ln), allocatable :: grd_names(:)          ! grid names
        character(len=max_str_ln), allocatable :: att_names(:)          ! attribute names

        ! set up local col_id and col
        col_id_loc = 4                                                  ! default collection dimension: last index
        if (present(col_id)) col_id_loc = col_id
        col_loc = 1                                                     ! default no spatial collection
        if (present(col)) col_loc = col

        ! set up nr. of plots
        n_plot = size(vars,col_id_loc)

        ! set up local tot_dim and loc_offset
        tot_dim_loc = shape(vars)
        if (present(tot_dim)) tot_dim_loc = tot_dim
        loc_offset_loc = 0
        if (present(loc_offset)) loc_offset_loc = loc_offset

        ! tests
        if (tot_dim_loc(col_id_loc).ne.n_plot) then
            istat = 1
            call writo('In plot_HDF5, all the processes need to have the full &
                &range in the dimension given by col_id',persistent=.true.,&
                &warning=.true.)
            chckstt
        end if
        if (n_plot.eq.1 .and. col_loc.ne.1) then
            istat = 1
            call writo('In plot_HDF5, if single plot, the collection type &
                &needs to be one',persistent=.true.,warning=.true.)
            chckstt
        end if

        ! set 3D dimensions
        col_mask = .false.
        col_mask(col_id_loc) = .true.
        tot_dim_3d = pack(tot_dim_loc,.not.col_mask)
        loc_dim_3d = pack(shape(vars),.not.col_mask)
        loc_offset_3d = pack(loc_offset_loc,.not.col_mask)

        ! set up individual plot
        if (tot_dim_3d(1).eq.loc_dim_3d(1) .and. &
            &tot_dim_3d(2).eq.loc_dim_3d(2) .and. &
            &tot_dim_3d(3).eq.loc_dim_3d(3)) then
            ind_plot = .true.
        else
            ind_plot = .false.
        end if

        ! set up continued plot
        cont_plot_loc = .false.
        if (present(cont_plot)) cont_plot_loc = cont_plot

        ! default symmetry type
        sym_type_loc = 1
```

```fortran

        ! Find  symmetry type  by  checking whether  Y/X  is constant  (toroidal
        ! symmetry) or  Z^2/(X^2+Y^2) is  constant (poloidal symmetry),  for all
        ! plots.
        if (present(sym_type)) then
            sym_type_loc = sym_type
        else if (minval(tot_dim_3d).eq.1) then                              ! possibly symmetry
            ! allocate helper variable
            allocate(sym_ang(loc_dim_3d(1),loc_dim_3d(2),loc_dim_3d(3)))
            ! initialize sym_pol and sym_tor
            sym_pol = 0
            sym_tor = 0
            ! loop over all plots
            do id = 1,n_plot
                ! assign pointers
                call assign_pointers(id)
                ! check poloidal angle
                sym_ang = atan2(y_3d,x_3d)
                if (maxval(sym_ang)-minval(sym_ang).lt.tol_sym .and. &
                    &(maxval(x_3d).ge.0._dp .neqv. &
                    &minval(x_3d).lt.0._dp).and.&                           ! X has to be either positive or negative
                    &(maxval(y_3d).ge.0._dp .neqv. &
                    &minval(y_3d).lt.0._dp)) &                              ! Y has to be either positive or negative
                    &sym_pol = sym_pol+1                                    ! poloidal symmetry for this plot
                ! check toroidal angle
                sym_ang = atan2(sqrt(z_3d**2),sqrt(x_3d**2+y_3d**2))
                if (maxval(sym_ang)-minval(sym_ang).lt.tol_sym) &
                    &sym_tor = sym_tor+1                                    ! toroidal symmetry for this plot
            end do
            ! get total sym_pol and sym_tor
            if (ind_plot) then                                              ! so that below test succeeds
                allocate(tot_sym_pol(n_procs),tot_sym_tor(n_procs))
                tot_sym_pol = sym_pol
                tot_sym_tor = sym_tor
            else                                                            ! get from all the processes
                istat = get_ser_var([sym_pol],tot_sym_pol,scatter=.true.)
                chckstt
                istat = get_ser_var([sym_tor],tot_sym_tor,scatter=.true.)
                chckstt
            end if

            ! check total results
            if (sum(tot_sym_pol).eq.n_procs*n_plot) then                    ! poloidal symmetry for all plots
                sym_type_loc = 2
            else if (sum(tot_sym_tor).eq.n_procs*n_plot) then               ! toroidal symmetry for all plots
                sym_type_loc = 3
            end if
            ! deallocate helper variables
            deallocate(sym_ang)
        end if

        ! tests
        if (col_loc.eq.4) then
            if (sym_type_loc.eq.1) then
                if (n_plot.ne.3) then
                    istat = 1
                    call writo('For vector field plots, need 3 dimensions',&
                        &warning=.true.)
                    chckstt
                end if
            else
                if (n_plot.ne.2) then
                    istat = 1
                    call writo('For symmetric vector field plots, need 2 &
                        &dimensions',warning=.true.)
                    chckstt
                end if
            end if
        end if

        ! set up and local var names if not a continued plot
        if (.not.cont_plot_loc) then
            ! set up local var_names
            allocate(grd_names(n_plot))
            allocate(att_names(n_plot))
            if (col_loc.eq.1) then                                          ! without collection
                if (n_plot.eq.1) then                                       ! just one plot: attribute name is important
                    if (size(var_names).eq.n_plot) then                     ! the right number of variable names provided
                        att_names = var_names
                    else if (size(var_names).gt.n_plot) then                ! too many variable names provided
                        att_names = var_names(1:n_plot)
                        call writo('Too many variable names provided',&
                            &persistent=.true.,warning=.true.)
                    else                                                    ! not enough variable names provided
                        att_names(1:size(var_names)) = var_names
                        do id = size(var_names)+1,n_plot
```

```
                        att_names(id) = 'unnamed variable '//trim(i2str(id))
                    end do
                    call writo('Not enough variable names provided',&
                        &persistent=.true.,warning=.true.)
                end if
                grd_names = 'default_grid_name'
            else                                                        ! multiple plots: grid name is important
                if (size(var_names).eq.n_plot) then                     ! the right number of variable names provided
                    grd_names = var_names
                else if (size(var_names).gt.n_plot) then                ! too many variable names provided
                    grd_names = var_names(1:n_plot)
                    call writo('Too many variable names provided',&
                        &persistent=.true.,warning=.true.)
                else                                                    ! not enough variable names provided
                    grd_names(1:size(var_names)) = var_names
                    do id = size(var_names)+1,n_plot
                        grd_names(id) = 'unnamed variable '//trim(i2str(id))
                    end do
                    call writo('Not enough variable names provided',&
                        &persistent=.true.,warning=.true.)
                end if
                att_names = 'default_att_name'
            end if
        else                                                            ! collections: attribute name is important
            att_names = var_names(1)
            if (size(var_names).gt.1) call writo('For collections, only &
                &the first variable name is used',persistent=.true.,&
                &warning=.true.)
            grd_names = 'default_grid_name'
        end if
    end if

    ! open HDF5 file
    istat = open_hdf5_file(file_info,file_name,sym_type=sym_type_loc,&
        &descr=descr,ind_plot=ind_plot,cont_plot=cont_plot_loc)
    chckstt

    ! create grid for collection if not continued plot
    if (.not.cont_plot_loc) allocate(grids(n_plot))

    ! allocate data item arrays
    if (sym_type_loc.eq.1) then                                         ! 3D grid
        allocate(dat(3))
    else                                                                ! 2D grid
        allocate(dat(2))
    end if

    ! loop over all plots
    do id = 1,n_plot
        ! set up whether last vector component
        ! (for scalar plots, this is always true)
        first_vc = (id.eq.1 .or. col_loc.ne.4)
        last_vc = (id.eq.n_plot .or. col_loc.ne.4)

        ! print topology if not continued plot and last vector components
        if (.not.cont_plot_loc .and. last_vc) then
            if (sym_type_loc.eq.1) then                                 ! 3D grid
                call print_hdf5_top(top,2,tot_dim_3d,ind_plot=ind_plot)
            else                                                        ! 2D grid
                call print_hdf5_top(top,1,tot_dim_3d,ind_plot=ind_plot)
            end if
        end if

        ! assign pointers
        call assign_pointers(id)

        ! print data  item for X, Y  and Z (no symmetry),  R = sqrt(X^2+Y^2)
        ! and Z (poloidal symmetry) or X  and Y (toroidal symmetry) if first
        ! vector component (afterwards, dat is overwritten)
        if (first_vc) then
            select case (sym_type_loc)
                case (1)                                                ! no symmetry
                    istat = print_hdf5_3d_data_item(dat(1),file_info,&
                        &'X_'//trim(i2str(id)),x_3d,tot_dim_3d,loc_dim_3d,&
                        &loc_offset_3d,ind_plot=ind_plot,&
                        &cont_plot=cont_plot_loc)
                    chckstt
                    istat = print_hdf5_3d_data_item(dat(2),file_info,&
                        &'Y_'//trim(i2str(id)),y_3d,tot_dim_3d,loc_dim_3d,&
                        &loc_offset_3d,ind_plot=ind_plot,&
                        &cont_plot=cont_plot_loc)
                    chckstt
                    istat = print_hdf5_3d_data_item(dat(3),file_info,&
                        &'Z_'//trim(i2str(id)),z_3d,tot_dim_3d,loc_dim_3d,&
                        &loc_offset_3d,ind_plot=ind_plot,&
                        &cont_plot=cont_plot_loc)
```

```
                    chckstt
                case (2)                                                ! poloidal symmetry
                    istat = print_hdf5_3d_data_item(dat(1),file_info,&
                        &'R_'//trim(i2str(id)),sqrt(x_3d**2+y_3d**2),&
                        &tot_dim_3d,loc_dim_3d,loc_offset_3d,&
                        &ind_plot=ind_plot,cont_plot=cont_plot_loc)
                    chckstt
                    istat = print_hdf5_3d_data_item(dat(2),file_info,&
                        &'Z_'//trim(i2str(id)),z_3d,tot_dim_3d,loc_dim_3d,&
                        &loc_offset_3d,ind_plot=ind_plot,&
                        &cont_plot=cont_plot_loc)
                    chckstt
                case (3)                                                ! toroidal symmetry
                    istat = print_hdf5_3d_data_item(dat(1),file_info,&
                        &'X_'//trim(i2str(id)),x_3d,tot_dim_3d,loc_dim_3d,&
                        &loc_offset_3d,ind_plot=ind_plot,&
                        &cont_plot=cont_plot_loc)
                    chckstt
                    istat = print_hdf5_3d_data_item(dat(2),file_info,&
                        &'Y_'//trim(i2str(id)),y_3d,tot_dim_3d,loc_dim_3d,&
                        &loc_offset_3d,ind_plot=ind_plot,&
                        &cont_plot=cont_plot_loc)
                    chckstt
                case default                                            ! no symmetry
                    istat = 1
                    call writo('symmetry type '//&
                        &trim(i2str(sym_type_loc))//' not recognized',&
                        &persistent=.true.,warning=.true.)
                    chckstt
            end select
        end if

        ! print geometry with X, Y and Z data item if not continued plot and
        ! first vector component
        if (.not.cont_plot_loc .and. first_vc) then
            if (sym_type_loc.eq.1) then                                 ! no symmetry so 3D geometry
                call print_hdf5_geom(geom,2,dat,reset=.true.,&
                    &ind_plot=ind_plot)
            else                                                        ! symmetry so 2D geometry
                call print_hdf5_geom(geom,1,dat,reset=.true.,&
                    &ind_plot=ind_plot)
            end if
        end if

        ! print data item for plot variable
        if (col_loc.eq.4) then
            att_id = id
        else
            att_id = 1
        end if
        istat = print_hdf5_3d_data_item(dat(att_id),file_info,'var_'//&
            &trim(i2str(id)),var_3d,tot_dim_3d,loc_dim_3d,&
            &loc_offset_3d,ind_plot=ind_plot,cont_plot=cont_plot_loc)
        chckstt

        ! print attribute with this data item if not continued plot
        if (.not.cont_plot_loc) then
            if (col_loc.eq.4) then
                if (last_vc) then
                    call merge_hdf5_3d_data_items(dat_vec,dat,'var_vec',&
                        &tot_dim_3d,reset=.true.,ind_plot=ind_plot,&
                        &cont_plot=cont_plot_loc)
                    call print_hdf5_att(att(1),dat_vec,&
                        &att_names(1),1,2,reset=.true.,ind_plot=ind_plot)
                end if
            else
                call print_hdf5_att(att(1),dat(1),&
                    &att_names(id),1,1,reset=.true.,ind_plot=ind_plot)
            end if
        end if

        ! create a grid  with the topology, the geometry,  the attribute and
        ! time if  time collection,  if not continued  plot and  last vector
        ! component
        if (.not.cont_plot_loc .and. last_vc) then
            if (col_loc.eq.2) then                                      ! time collection
                istat = print_hdf5_grid(grids(id),grd_names(id),1,&
                    &grid_time=id*1._dp,grid_top=top,grid_geom=geom,&
                    &grid_atts=att,reset=.true.,ind_plot=ind_plot)
                chckstt
            else                                                        ! no time collection
                istat = print_hdf5_grid(grids(id),grd_names(id),1,&
                    &grid_top=top,grid_geom=geom,grid_atts=att,&
                    &reset=.true.,ind_plot=ind_plot)
                chckstt
            end if
        end if
```

```
        end do

        ! either create collection or just use individual grids if not continued
        ! plot
        if (.not.cont_plot_loc) then
            if (col_loc.eq.1 .or. col_loc.eq.4) then
                ! add individual grids to HDF5 file and reset them
                do id = 1,n_plot
                    if (col_loc.eq.4 .and. id.lt.n_plot) cycle                  ! only plot last for vector
                    istat = add_hdf5_item(file_info,grids(id),reset=.true.,&
                    &ind_plot=ind_plot)
                    chckstt
                end do
            else
                ! create grid collection from individual grids and reset them
                istat = print_hdf5_grid(col_grid,'domain of mesh',col_loc,&
                    &grid_grids=grids,reset=.true.,ind_plot=ind_plot)
                chckstt

                ! add collection grid to HDF5 file and reset it
                istat = add_hdf5_item(file_info,col_grid,reset=.true.,&
                    &ind_plot=ind_plot)
                chckstt
            end if
        end if

        ! close HDF5 file
        istat = close_hdf5_file(file_info,ind_plot=ind_plot,&
            &cont_plot=cont_plot_loc)
        chckstt

        ! clean up
        nullify(var_3d)
        nullify(x_3d,y_3d,z_3d)
        call dealloc_xml_str(dat)
        if (.not.cont_plot_loc) then
            call dealloc_xml_str(grids)
            call dealloc_xml_str(top)
            call dealloc_xml_str(geom)
            call dealloc_xml_str(att(1))
            call dealloc_xml_str(col_grid)
        end if
    contains
        !> \private assigns the 3D subarray pointer variables
        subroutine assign_pointers(id)
            ! input / output
            integer :: id                                                       ! index at which to assing pointer

            ! local variables
            integer :: id_loc                                                   ! local id

            ! X
            if (present(x)) then
                id_loc = id
                if (col_id_loc.eq.1) then
                    if (size(x,1).eq.1) id_loc = 1
                    x_3d => x(id_loc,:,:,:)
                else if (col_id_loc.eq.2) then
                    if (size(x,2).eq.1) id_loc = 1
                    x_3d => x(:,id_loc,:,:)
                else if (col_id_loc.eq.3) then
                    if (size(x,3).eq.1) id_loc = 1
                    x_3d => x(:,:,id_loc,:)
                else if (col_id_loc.eq.4) then
                    if (size(x,4).eq.1) id_loc = 1
                    x_3d => x(:,:,:,id_loc)
                else
                    istat = 1
                    chckstt
                end if
            else
                allocate(x_3d(loc_dim_3d(1),loc_dim_3d(2),loc_dim_3d(3)))
                do jd = 1,loc_dim_3d(1)
                    x_3d(jd,:,:) = loc_offset_3d(1) + jd - 1
                end do
            end if
            ! Y
            if (present(y)) then
                id_loc = id
                if (col_id_loc.eq.1) then
                    if (size(y,1).eq.1) id_loc = 1
                    y_3d => y(id_loc,:,:,:)
                else if (col_id_loc.eq.2) then
                    if (size(y,2).eq.1) id_loc = 1
                    y_3d => y(:,id_loc,:,:)
                else if (col_id_loc.eq.3) then
                    if (size(y,3).eq.1) id_loc = 1
```

```
                    y_3d => y(:,:,id_loc,:)
                else if (col_id_loc.eq.4) then
                    if (size(y,4).eq.1) id_loc = 1
                    y_3d => y(:,:,:,id_loc)
                else
                    istat = 1
                    chckstt
                end if
            else
                allocate(y_3d(loc_dim_3d(1),loc_dim_3d(2),loc_dim_3d(3)))
                do jd = 1,loc_dim_3d(2)
                    y_3d(:,jd,:) = loc_offset_3d(2) + jd - 1
                end do
            end if
            ! Z
            if (present(z)) then
                id_loc = id
                if (col_id_loc.eq.1) then
                    if (size(z,1).eq.1) id_loc = 1
                    z_3d => z(id_loc,:,:,:)
                else if (col_id_loc.eq.2) then
                    if (size(z,2).eq.1) id_loc = 1
                    z_3d => z(:,id_loc,:,:)
                else if (col_id_loc.eq.3) then
                    if (size(z,3).eq.1) id_loc = 1
                    z_3d => z(:,:,id_loc,:)
                else if (col_id_loc.eq.4) then
                    if (size(z,4).eq.1) id_loc = 1
                    z_3d => z(:,:,:,id_loc)
                else
                    istat = 1
                    chckstt
                end if
            else
                allocate(z_3d(loc_dim_3d(1),loc_dim_3d(2),loc_dim_3d(3)))
                do jd = 1,loc_dim_3d(3)
                    z_3d(:,:,jd) = loc_offset_3d(3) + jd - 1
                end do
            end if
            ! variable
            if (col_id_loc.eq.1) then
                var_3d => vars(id,:,:,:)
            else if (col_id_loc.eq.2) then
                var_3d => vars(:,id,:,:)
            else if (col_id_loc.eq.3) then
                var_3d => vars(:,:,id,:)
            else if (col_id_loc.eq.4) then
                var_3d => vars(:,:,:,id)
            else
                istat = 1
                chckstt
            end if
        end subroutine assign_pointers
    end subroutine plot_hdf5_arr
    !> \private individual version
    subroutine plot_hdf5_ind(var_name,file_name,var,tot_dim,loc_offset,&
        &X,Y,Z,cont_plot,descr)

        ! input / output
        character(len=*), intent(in) :: var_name                                !< name of variable to be plot
        character(len=*), intent(in) :: file_name                               !< file name
        real(dp), intent(in) :: var(:,:,:)                                      !< variable to plot
        integer, intent(in), optional :: tot_dim(3)                             !< total dimensions of the arrays
        integer, intent(in), optional :: loc_offset(3)                          !< offset of local dimensions
        real(dp), intent(in), target, optional :: X(:,:,:)                      !< curvilinear grid X points
        real(dp), intent(in), target, optional :: Y(:,:,:)                      !< curvilinear grid Y points
        real(dp), intent(in), target, optional :: Z(:,:,:)                      !< curvilinear grid Z points
        logical, intent(in), optional :: cont_plot                              !< continued plot
        character(len=*), intent(in), optional :: descr                         !< description

        ! local variables
        integer :: tot_dim_loc(4), loc_offset_loc(4)                            ! local versions of total dimensions and local offset

        ! set up local tot_dim and loc_offset
        tot_dim_loc = [shape(var),1]
        if (present(tot_dim)) tot_dim_loc = [tot_dim,1]
        loc_offset_loc = 0
        if (present(loc_offset)) loc_offset_loc = [loc_offset,0]

        if (present(x)) then
            if (present(y)) then
                if (present(z)) then
                    call plot_hdf5_arr([var_name],file_name,&
                        &reshape(var,[size(var,1),size(var,2),size(var,3),1]),&
                        &tot_dim_loc,loc_offset_loc,&
                        &x=reshape(x,[size(x,1),size(x,2),size(x,3),1]),&
```

```
                        &y=reshape(y,[size(y,1),size(y,2),size(y,3),1]),&
                        &z=reshape(z,[size(z,1),size(z,2),size(z,3),1]),&
                        &col=1,cont_plot=cont_plot,descr=descr)
                else
                    call plot_hdf5_arr([var_name],file_name,&
                        &reshape(var,[size(var,1),size(var,2),size(var,3),1]),&
                        &tot_dim_loc,loc_offset_loc,&
                        &x=reshape(x,[size(x,1),size(x,2),size(x,3),1]),&
                        &y=reshape(y,[size(y,1),size(y,2),size(y,3),1]),&
                        &col=1,cont_plot=cont_plot,descr=descr)
                end if
            else
                if (present(z)) then
                    call plot_hdf5_arr([var_name],file_name,&
                        &reshape(var,[size(var,1),size(var,2),size(var,3),1]),&
                        &tot_dim_loc,loc_offset_loc,&
                        &x=reshape(x,[size(x,1),size(x,2),size(x,3),1]),&
                        &z=reshape(z,[size(z,1),size(z,2),size(z,3),1]),&
                        &col=1,cont_plot=cont_plot,descr=descr)
                else
                    call plot_hdf5_arr([var_name],file_name,&
                        &reshape(var,[size(var,1),size(var,2),size(var,3),1]),&
                        &tot_dim_loc,loc_offset_loc,&
                        &x=reshape(x,[size(x,1),size(x,2),size(x,3),1]),&
                        &col=1,cont_plot=cont_plot,descr=descr)
                end if
            end if
        else
            if (present(y)) then
                if (present(z)) then
                    call plot_hdf5_arr([var_name],file_name,&
                        &reshape(var,[size(var,1),size(var,2),size(var,3),1]),&
                        &tot_dim_loc,loc_offset_loc,&
                        &y=reshape(y,[size(y,1),size(y,2),size(y,3),1]),&
                        &z=reshape(z,[size(z,1),size(z,2),size(z,3),1]),&
                        &col=1,cont_plot=cont_plot,descr=descr)
                else
                    call plot_hdf5_arr([var_name],file_name,&
                        &reshape(var,[size(var,1),size(var,2),size(var,3),1]),&
                        &tot_dim_loc,loc_offset_loc,&
                        &y=reshape(y,[size(y,1),size(y,2),size(y,3),1]),&
                        &col=1,cont_plot=cont_plot,descr=descr)
                end if
            else
                if (present(z)) then
                    call plot_hdf5_arr([var_name],file_name,&
                        &reshape(var,[size(var,1),size(var,2),size(var,3),1]),&
                        &tot_dim_loc,loc_offset_loc,&
                        &col=1,z=reshape(z,[size(z,1),size(z,2),size(z,3),1]),&
                        &cont_plot=cont_plot,descr=descr)
                else
                    call plot_hdf5_arr([var_name],file_name,&
                        &reshape(var,[size(var,1),size(var,2),size(var,3),1]),&
                        &tot_dim_loc,loc_offset_loc,&
                        &col=1,cont_plot=cont_plot,descr=descr)
                end if
            end if
        end if
    end subroutine plot_hdf5_ind

    !> Use external program to draw a plot.
    !!
    !!
    !! The external programs used are determined by \c ex_plot_style:
    !!  - \c ex_plot_style = 1: Use GnuPlot for 2-D and 3-D
    !!  - \c ex_plot_style = 1: Use Bokeh in 2-D and Mayavi in 3-D
    !!
    !! The  output  is saved  in  a  file <tt>[draw_name].[ext]</tt>  where  the
    !! extension depends  on the  plotting style  and the  dimension, or  on the
    !! screen, depending on \c plot_on_screen.
    !!
    !! If  not specified  otherwise through  data_name, it  is assumed  that the
    !! datafile to be plot is situated in <tt>[draw_name].dat</tt>.
    !!
    !! The title(s)  of the plot(s)  are provided  through \c var_names  and the
    !! number of plots through \c nplt. If  there are less names than the number
    !! of plots, the first one is taken and appended by the plot number.
    !!
    !! Furthermore, \c draw_dim determines whether the plot is to be 2-D, 3-D or
    !! 2-D slices in 3-D.
    !!
    !! Also, an optional command \c extra_ops can be provided, to provide overal
    !! options, as well as draw_ops that  specifies the line style for the plots
    !! from the file. If \c less draw_ops  are provided than plots, they will be
    !! cycled.
    !!
    !! Finally,  there is  an  option to  provide animated  plots,  but is  only
```

```
    !! available in 2-D.  Also, for external plot style 1,  no on-screen view is
    !! possible. Animations can take optional ranges arguments as well as delay.
    !! \note \c About draw_dim:
    !!  -  \c draw_dim  = 1:  2-D  plot; should  be  called with  the output  of
    !!  output_ops.print_ex_2d(), \c nplt should be correctly set.
    !!  -  \c draw_dim  = 2:  3-D  plot; should  be  called with  the output  of
    !!  print_ex_3D. For GNUPlot, \c nplt is  ignored, but not for Mayavi, as it
    !!  needs this information to be able to  reconstruct 2D arrays for X, Y and
    !!  Z.
    !!  - \c  draw_dim = 3: 2-D  plot in 3-D  slices; should be called  with the
    !!  output of output_ops.print_ex_2d(), \c nplt should be correctly set.
    subroutine draw_ex(var_names,draw_name,nplt,draw_dim,plot_on_screen,&
        &ex_plot_style,data_name,draw_ops,extra_ops,is_animated,ranges,delay,&
        &persistent)

        use num_vars, only: ex_plot_style_global => ex_plot_style, rank

        ! input / output
        character(len=*), intent(in) :: var_names(:)                          !< name of variables
        character(len=*), intent(in) :: draw_name                             !< name of drawing
        integer, intent(in) :: nplt                                           !< number of plots
        integer, intent(in) :: draw_dim                                       !< 1: 2-D, 2: 3-D, 3: decoupled 3-D
        logical, intent(in) :: plot_on_screen                                 !< True if on screen, false if in file
        integer, intent(in), optional :: ex_plot_style                        !< alternative external plot style
        character(len=*), intent(in), optional :: data_name                   !< name of data file
        character(len=*), intent(in), optional :: draw_ops(:)                 !< drawing options
        character(len=*), intent(in), optional :: extra_ops                   !< extra options
        logical, intent(in), optional :: is_animated                          !< plot is animated
        real(dp), intent(in), optional :: ranges(:,:)                         !< x and y range of animated plot
        integer, intent(in), optional :: delay                                !< time delay between animated plot frames
        logical, intent(in), optional :: persistent                           !< keep on-screen plot open

        ! local variables
        character(len=4) :: ex_ext                                            ! output extension of external program
        character(len=5*max_str_ln) :: cmdmsg                                 ! error message of C system command
        character(len=max_str_ln) :: script_name                              ! name of script, including path
        character(len=max_str_ln) :: ex_prog_name                             ! name of external program
        character(len=max_str_ln) :: data_name_loc                            ! local data name
        character(len=max_str_ln), allocatable :: var_names_loc(:)            ! local variables names
        logical :: is_animated_loc                                            ! local is_animated
        logical :: run_shell_necessary                                        ! whether shell needs to be run
        logical :: persistent_loc                                             ! local persistent
        real(dp), allocatable :: ranges_loc(:,:)                              ! local copy of ranges
        integer :: delay_loc                                                  ! local copy of delay
        integer :: n_draw_ops                                                 ! number of draw options provided
        integer :: istat                                                      ! status of opening a file
        integer :: iplt                                                       ! counter
        integer :: cmd_i                                                      ! file number for script file
        integer :: cmdstat                                                    ! status of C system command
        integer :: ex_plot_style_loc                                          ! local ex_plot_style

        ! skip if no plots
        if (no_plots) return

        ! set up local data_name, variable names, is_animated and ex_plot_style
        data_name_loc = draw_name
        if (present(data_name)) data_name_loc = data_name
        allocate(var_names_loc(nplt))
        if (size(var_names).eq.nplt) then
            var_names_loc = var_names
        else
            do iplt = 1,nplt
                var_names_loc(iplt) = trim(var_names(1))//' ('//&
                    &trim(i2str(iplt))//' / '//trim(i2str(nplt))//')'
            end do
        end if
        is_animated_loc = .false.
        if (present(is_animated)) is_animated_loc = is_animated
        ex_plot_style_loc = ex_plot_style_global
        if (present(ex_plot_style)) ex_plot_style_loc = ex_plot_style
        persistent_loc = .false.
        if (present(persistent)) persistent_loc = persistent

        ! run shell to produce the plot by default
        run_shell_necessary = .true.

        ! test whether animation is in 2D and set some variables
        if (is_animated_loc) then
            if (draw_dim.ne.1) then
                call writo('Animations are only possible in 2D',warning=.true.,&
                    &persistent=.true.)
                return
            else
                ! calculate delay [1/100 s]
                if (present(delay)) then
                    delay_loc = delay
                else
```

```
                delay_loc = max(250/nplt,10)
            end if
        end if
    end if

    ! create the external program command
    if (plot_on_screen .and. &
        &(ex_plot_style_loc.ne.1 .and. is_animated_loc)) then          ! not for GNUPlot animation
        script_name = trim(script_dir)//'/'//'temp_script_draw_ex_'//&
            &trim(i2str(rank))//'_'//trim(i2str(temp_id(2)))
        temp_id(2) = temp_id(2)+1
    else
        script_name = trim(script_dir)//'/'//trim(draw_name)
    end if
    select case (ex_plot_style_loc)
        case (1)                                                       ! GNUPlot
            script_name = trim(script_name)//'.gnu'
            ex_ext = 'pdf'
            ex_prog_name = 'gnuplot'
        case (2)                                                       ! Bokeh or Mayavi
            script_name = trim(script_name)//'.py'
            if (draw_dim.eq.1) then                                    ! 2D: Bokeh
                ex_ext = 'html'
            else                                                       ! 3D and 2D slices in 3D: Mayavi
                ex_ext = 'png'
            end if
            ex_prog_name = 'python'
    end select

    ! open script file
    open(nextunit(cmd_i),file=trim(script_name),iostat=istat)
    chckstt

    ! set number of draw ops
    if (present(draw_ops)) then
        n_draw_ops = size(draw_ops)
    else
        n_draw_ops = 0
    end if

    select case (ex_plot_style_loc)
        case (1)                                                       ! GNUPlot
            if (is_animated_loc) then
                call draw_ex_animated_gnuplot()
            else
                call draw_ex_gnuplot()
            end if
        case (2)                                                       ! Bokeh
            select case (draw_dim)
                case (1)                                               ! 2D
                    !!run_shell_necessary = plot_on_screen              ! only need to run if plot on screen (?)
                    if (is_animated_loc) then
                        call draw_ex_animated_bokeh()
                    else
                        call draw_ex_bokeh()
                    end if
                case (2:3)                                             ! 3D, 2D slices in 3D
                    call draw_ex_mayavi()
            end select
    end select

    ! closing the command
    close(cmd_i)

    ! run shell if necessary
    istat = 0
    cmdstat = 0
    if (run_shell_necessary) &
        &call use_execute_command_line(trim(ex_prog_name)//' "'//&
            &trim(script_name)//'"'//err_output_str,exitstat=istat,&
            &cmdstat=cmdstat,cmdmsg=cmdmsg)

    if (istat.ne.0) then
        call writo('Failed to plot '//trim(draw_name)//'.'//&
            &trim(ex_ext),persistent=.true.,warning=.true.)
    else
        if (cmdstat.ne.0) then
            call writo('Failed to plot '//trim(draw_name)//'.'//&
                &trim(ex_ext),persistent=.true.,warning=.true.)
            call lvl_ud(1)
            call writo('System message: "'//trim(cmdmsg)//'"',&
                &persistent=.true.)
            if (.not.plot_on_screen) call writo(&
                &'Try running "'//trim(ex_prog_name)//' "'//&
                &trim(script_name)//'"'//'" manually',persistent=.true.)
            call lvl_ud(-1)
        else
```

```
                if (plot_on_screen .and. run_shell_necessary .and. &
                    &.not.persistent_loc) then
                    call use_execute_command_line('rm '//&
                        &trim(script_name),exitstat=istat,&
                        &cmdstat=cmdstat,cmdmsg=cmdmsg)
                    ! ignore errors
                else
                    call writo('Plot in output file "'//&
                        &trim(plot_dir)//'/'//trim(draw_name)//&
                        &'.'//trim(ex_ext)//'"',persistent=.true.)
                end if
            end if
        end if
    contains
        !> \private GNUPlot version: wxt terminal or pdf output
        subroutine draw_ex_gnuplot
            ! initialize the script
            write(cmd_i,"(A)",iostat=istat) 'set grid'
            write(cmd_i,"(A)",iostat=istat) 'set border 4095 front linetype -1 &
                &linewidth 1.0'
            if (plot_on_screen) then
                if (persistent_loc) then
                    write(cmd_i,"(A)",iostat=istat) 'set terminal wxt persist'
                else
                    write(cmd_i,"(A)",iostat=istat) 'set terminal wxt'
                end if
            else
                write(cmd_i,"(A)",iostat=istat) 'set terminal pdf size '//&
                    &trim(i2str(plot_size(1)))//','//trim(i2str(plot_size(2)))
                write(cmd_i,"(A)",iostat=istat) 'set output "'//&
                    &trim(plot_dir)//'/'//trim(draw_name)//'.pdf"'
            end if

            ! no legend if too many plots
            if (nplt.gt.10) write(cmd_i,"(A)",iostat=istat) 'set nokey;'

            ! write extra options
            if (present(extra_ops)) write(cmd_i,"(A)",iostat=istat) &
                &trim(extra_ops)

            ! set up line styles
            if (n_draw_ops.eq.0) then
                do iplt = 1,min(nplt,size(line_clrs))
                    write(cmd_i,"(A)",iostat=istat) 'set style line '//&
                        &trim(i2str(iplt))//' lc rgb '//&
                        &line_clrs(iplt)//' '//trim(line_style)
                end do
            end if

            ! individual plots
            select case (draw_dim)
                case (1)                                                ! 2D
                    write(cmd_i,"(A)",iostat=istat) 'plot \'
                    do iplt = 1,nplt
                        write(cmd_i,"(A)",iostat=istat,advance="no") &
                            &' "'//trim(data_dir)//'/'//trim(data_name_loc)//&
                            &'.dat" using '//trim(i2str(iplt))//':'//&
                            &trim(i2str(nplt+iplt))//' title "'//&
                            &trim(var_names_loc(iplt))//'" '//&
                            &trim(loc_draw_op())
                        if (iplt.eq.nplt) then
                            write(cmd_i,"(A)",iostat=istat) ''
                        else
                            write(cmd_i,"(A)",iostat=istat) ', \'
                        end if
                    end do
                case (2)                                                ! 3D
                    write(cmd_i,"(A)",iostat=istat) 'splot \'
                    do iplt = 1,nplt
                        write(cmd_i,"(A)",iostat=istat,advance="no") &
                            &' "'//trim(data_dir)//'/'//trim(data_name_loc)//&
                            &'.dat" using '//trim(i2str(iplt))//':'//&
                            &trim(i2str(nplt+iplt))//':'//&
                            &trim(i2str(2*nplt+iplt))//' title "'//&
                            &trim(var_names_loc(iplt))//&
                            &'" '//trim(loc_draw_op())
                        if (iplt.eq.nplt) then
                            write(cmd_i,"(A)",iostat=istat) ''
                        else
                            write(cmd_i,"(A)",iostat=istat) ', \'
                        end if
                    end do
                case (3)                                                ! 2D slices in 3D
                    write(cmd_i,"(A)",iostat=istat) 'splot \'
                    do iplt = 1,nplt
                        write(cmd_i,"(A)",iostat=istat,advance="no") &
                            &' "'//trim(data_dir)//'/'//trim(data_name_loc)//&
```

```
                        &'.dat" using ('//trim(i2str(iplt))//'):'//&
                        &trim(i2str(iplt))//':'//trim(i2str(nplt+iplt))//&
                        &' title "'//trim(var_names_loc(iplt))//'" '//&
                        &trim(loc_draw_op())
                    if (iplt.eq.nplt) then
                        write(cmd_i,"(A)",iostat=istat) ''
                    else
                        write(cmd_i,"(A)",iostat=istat) ', \'
                    end if
                end do
            case default
                call writo('No draw_dim associated with '//&
                    &trim(i2str(draw_dim)),persistent=.true.)
                istat = 1
                chckstt
        end select

        ! finishing the command
        write(cmd_i,"(A)",iostat=istat) ''
        if (plot_on_screen) write(cmd_i,"(A)",iostat=istat) 'pause -1'
    end subroutine draw_ex_gnuplot

    !> \private Bokeh version: 2D html output
    ! Interactive checkbox from https://github.com/bokeh/bokeh/issues/3715.
    ! (Could be replaced by http://bokeh.pydata.org/en/latest/docs/
    !  user_guide/interaction/legends.html#userguide-interaction-legends)
    subroutine draw_ex_bokeh
        ! initialize the script
        write(cmd_i,"(A)",iostat=istat) 'from numpy import genfromtxt'
        write(cmd_i,"(A)",iostat=istat) 'from bokeh.plotting import &
            &figure, output_file, save, show'
        write(cmd_i,"(A)",iostat=istat) 'from bokeh.layouts import &
            &widgetbox, row'
        write(cmd_i,"(A)",iostat=istat) 'from bokeh.models.widgets import &
            &CheckboxGroup'
        write(cmd_i,"(A)",iostat=istat) 'from bokeh.models import CustomJS'
        write(cmd_i,"(A)",iostat=istat) ''
        write(cmd_i,"(A)",iostat=istat) 'TOOLS="hover,pan,wheel_zoom,&
            &box_zoom,reset,save"'
        write(cmd_i,"(A)",iostat=istat) ''
        write(cmd_i,"(A)",iostat=istat) 'output_file("'//trim(plot_dir)//&
            &'/'//trim(draw_name)//'.html", title="'//trim(draw_name)//&
            &'", mode="cdn") '
        write(cmd_i,"(A)",iostat=istat) ''
        write(cmd_i,"(A)",iostat=istat) 'data = genfromtxt("'//&
            &trim(data_dir)//'/'//trim(data_name_loc)//'.dat")'
        write(cmd_i,"(A)",iostat=istat) ''
        write(cmd_i,"(A)",iostat=istat) 'p = figure(toolbar_location=&
            &"above",tools=TOOLS,active_drag="box_zoom",active_scroll=&
            &"wheel_zoom",plot_width=600, plot_height=600)'
        write(cmd_i,"(A)",iostat=istat) ''

        ! write extra options
        if (present(extra_ops)) write(cmd_i,"(A)",iostat=istat) &
            &trim(extra_ops)

        ! individual plots
        do iplt = 1,nplt
            ! plot the lines
            write(cmd_i,"(A)",iostat=istat) 'l'//trim(i2str(iplt))//&
                &' = p.line(data[:,'//trim(i2str(iplt-1))//'],data[:,'//&
                &trim(i2str(nplt+iplt-1))//'],'//trim(loc_draw_op(1))//')'
            ! plot the circles
            write(cmd_i,"(A)",iostat=istat) 'c'//trim(i2str(iplt))//&
                &' = p.circle(data[:,'//&
                &trim(i2str(iplt-1))//'],data[:,'//&
                &trim(i2str(nplt+iplt-1))//'],'//trim(loc_draw_op(2))//')'
        end do
        write(cmd_i,"(A)",iostat=istat) ''

        ! create checkbox
        write(cmd_i,"(A)",iostat=istat) 'labels=list()'
        do iplt = 1,nplt
            write(cmd_i,"(A)",iostat=istat) 'labels.append("'//&
                &trim(var_names_loc(iplt))//'")'
        end do
        write(cmd_i,"(A)",iostat=istat) 'active=list()'
        do iplt = 1,nplt
            write(cmd_i,"(A)",iostat=istat) 'active.append('//&
                &trim(i2str(iplt-1))//')'
        end do
        write(cmd_i,"(A)",iostat=istat) 'checkbox = CheckboxGroup(&
            &labels=labels, active=active)'
        write(cmd_i,"(A)",iostat=istat) ''

        ! create callback
        write(cmd_i,"(A)",iostat=istat) 'args=dict()'
```

```fortran
            do iplt = 1,nplt
                write(cmd_i,"(A)",iostat=istat) 'args["l'//&
                    &trim(i2str(iplt))//'"] = l'//trim(i2str(iplt))
                write(cmd_i,"(A)",iostat=istat) 'args["c'//&
                    &trim(i2str(iplt))//'"] = c'//trim(i2str(iplt))
            end do

            write(cmd_i,"(A)",iostat=istat) 'code="""'
            write(cmd_i,"(A)",iostat=istat) '//console.log(cb_obj.active);'
            do iplt = 1,nplt
                write(cmd_i,"(A)",iostat=istat) 'l'//trim(i2str(iplt))//&
                    &'.visible = false;'
                write(cmd_i,"(A)",iostat=istat) 'c'//trim(i2str(iplt))//&
                    &'.visible = false;'
            end do
            write(cmd_i,"(A)",iostat=istat) 'for (i in cb_obj.active) {'
            write(cmd_i,"(A)",iostat=istat) &
                &'    //console.log(cb_obj.active[i]);'
            do iplt = 1,nplt
                write(cmd_i,"(A)",iostat=istat) '    if (cb_obj.active[i] == '&
                    &//trim(i2str(iplt-1))//') {'
                write(cmd_i,"(A)",iostat=istat) '    l'//trim(i2str(iplt))//&
                    &'.visible = true;'
                write(cmd_i,"(A)",iostat=istat) '    c'//trim(i2str(iplt))//&
                    &'.visible = true;'
                write(cmd_i,"(A)",iostat=istat) '    }'
            end do
            write(cmd_i,"(A)",iostat=istat) '}'
            write(cmd_i,"(A)",iostat=istat) '"""'

            write(cmd_i,"(A)",iostat=istat) 'checkbox.callback = CustomJS(&
                &args=args, code=code)'

            ! create layout
            write(cmd_i,"(A)",iostat=istat) &
                &'layout = row(p, widgetbox(checkbox))'
            write(cmd_i,"(A)",iostat=istat) ''

            ! finishing the command
            write(cmd_i,"(A)",iostat=istat) ''
            if (plot_on_screen) then
                write(cmd_i,"(A)",iostat=istat) 'show(layout)'
            else
                write(cmd_i,"(A)",iostat=istat) 'save(layout)'
            end if
        end subroutine draw_ex_bokeh

        !> \private Mayavi version: 3D png output
        subroutine draw_ex_mayavi
            ! initialize the script
            write(cmd_i,"(A)",iostat=istat) 'from numpy import genfromtxt, &
                &array, size, zeros, amin, amax'
            write(cmd_i,"(A)",iostat=istat) 'from mayavi.mlab import outline, &
                &mesh, points3d, savefig, colorbar, axes, savefig, show'
            write(cmd_i,"(A)",iostat=istat) ''
            write(cmd_i,"(A)",iostat=istat) 'nplt = '//trim(i2str(nplt))

            ! calculate the number of y points
            write(cmd_i,"(A)",iostat=istat) 'npnty = -1'
            write(cmd_i,"(A)",iostat=istat) 'with open("'//trim(data_dir)//&
                &'/'//trim(data_name_loc)//'.dat") as f:'
            write(cmd_i,"(A)",iostat=istat) '    for line in f:'
            write(cmd_i,"(A)",iostat=istat) '        if (npnty < 0):'
            write(cmd_i,"(A)",iostat=istat) '            if &
                &(line.strip()[0] == "#"):'
            write(cmd_i,"(A)",iostat=istat) '                npnty = 0'
            write(cmd_i,"(A)",iostat=istat) '        else:'
            write(cmd_i,"(A)",iostat=istat) '            if (not line.strip()):'
            write(cmd_i,"(A)",iostat=istat) '                break'
            write(cmd_i,"(A)",iostat=istat) '            else:'
            write(cmd_i,"(A)",iostat=istat) '                npnty += 1'

            ! get data
            write(cmd_i,"(A)",iostat=istat) 'data = genfromtxt("'//&
                &trim(data_dir)//'/'//trim(data_name_loc)//'.dat")'
            write(cmd_i,"(A)",iostat=istat) 'dims = &
                &array([size(data,0)/npnty,npnty,nplt])'
            write(cmd_i,"(A)",iostat=istat) ''
            write(cmd_i,"(A)",iostat=istat) 'X = zeros(dims)'
            write(cmd_i,"(A)",iostat=istat) 'Y = zeros(dims)'
            write(cmd_i,"(A)",iostat=istat) 'Z = zeros(dims)'
            write(cmd_i,"(A)",iostat=istat) 'for i in range(0,dims[0]):'
            write(cmd_i,"(A)",iostat=istat) '    for j in range(0,dims[2]):'
            write(cmd_i,"(A)",iostat=istat) '        X[i,:,j] = &
                &data[npnty*i:npnty*(i+1),j]'
            write(cmd_i,"(A)",iostat=istat) '        Y[i,:,j] = &
                &data[npnty*i:npnty*(i+1),nplt+j]'
```

```
            write(cmd_i,"(A)",iostat=istat) '        Z[i,:,j] = &
                &data[npnty*i:npnty*(i+1),nplt*2+j]'
            write(cmd_i,"(A)",iostat=istat) ''
            write(cmd_i,"(A)",iostat=istat) 'minZ = amin(Z)'
            write(cmd_i,"(A)",iostat=istat) 'maxZ = amax(Z)'
            write(cmd_i,"(A)",iostat=istat) ''

            ! write extra options
            if (present(extra_ops)) write(cmd_i,"(A)",iostat=istat) &
                &trim(extra_ops)

            ! plot
            select case (draw_dim)
                case (2)                                                    ! 3D
                    write(cmd_i,"(A)",iostat=istat) 'for j in range(0,dims[2]):'
                    write(cmd_i,"(A)",iostat=istat) '    &
                        &mesh(X[:,:,j],Y[:,:,j],Z[:,:,j],vmin=minZ,vmax=maxZ)'
                case (3)                                                    ! 2D slices in 3D
                    write(cmd_i,"(A)",iostat=istat) 'for j in range(0,dims[2]):'
                    write(cmd_i,"(A)",iostat=istat) '    &
                        &points3d(X[:,:,j],Y[:,:,j],Z[:,:,j],mode="sphere",&
                        &scale_factor=0.05,vmin=minZ,vmax=maxZ)'
            end select
            write(cmd_i,"(A)",iostat=istat) 'outline()'
            write(cmd_i,"(A)",iostat=istat) 'colorbar()'
            write(cmd_i,"(A)",iostat=istat) 'axes()'

            ! pause on screen or output to file
            if (plot_on_screen) then
                write(cmd_i,"(A)",iostat=istat) ''
                write(cmd_i,"(A)",iostat=istat) 'show()'
            else
                write(cmd_i,"(A)",iostat=istat) ''
                write(cmd_i,"(A)",iostat=istat) '#show()'
                write(cmd_i,"(A)",iostat=istat) 'savefig("'//trim(plot_dir)//&
                    &'/'//trim(draw_name)//'.png"'//',magnification=4)'
            end if
        end subroutine draw_ex_mayavi

        !> \private GNUPlot animated version: gif output
        subroutine draw_ex_animated_gnuplot
            ! initialize the script
            write(cmd_i,"(A)",iostat=istat) 'set grid'
            write(cmd_i,"(A)",iostat=istat) 'set border 4095 front linetype -1 &
                &linewidth 1.0'
            write(cmd_i,"(A)",iostat=istat) 'set terminal gif animate &
                &delay '//trim(i2str(delay_loc))//' size '//&
                &trim(i2str(128*plot_size(1)))//','//&
                &trim(i2str(128*plot_size(2)))
            write(cmd_i,"(A)",iostat=istat) 'set output "'//trim(plot_dir)//&
                &'/'//trim(draw_name)//'.gif"'

            ! find and set ranges
            allocate(ranges_loc(2,2))
            if (present(ranges)) then
                if (size(ranges,1).eq.2 .and. size(ranges,2).eq.2) then
                    ranges_loc = ranges
                else
                    call writo('invalid ranges given &
                        &to draw_ex_animated',persistent=.true.,&
                        &warning=.true.)
                end if
            else
                ! initialize ranges
                ranges_loc(:,1) = huge(1._dp)                               ! minimum value
                ranges_loc(:,2) = -huge(1._dp)                              ! maximum value

                call get_ranges(ranges_loc)
            end if

            ! set ranges
            write(cmd_i,"(A)",iostat=istat) 'set xrange ['//&
                &trim(r2str(ranges_loc(1,1)))//':'//&
                &trim(r2str(ranges_loc(1,2)))//'];'
            write(cmd_i,"(A)",iostat=istat) 'set yrange ['//&
                &trim(r2str(ranges_loc(2,1)))//':'//&
                &trim(r2str(ranges_loc(2,2)))//'];'

            ! write extra options
            if (present(extra_ops)) write(cmd_i,"(A)",iostat=istat) &
                &trim(extra_ops)

            ! set up line styles
            if (n_draw_ops.eq.0) then
                do iplt = 1,min(nplt,size(line_clrs))
                    write(cmd_i,"(A)",iostat=istat) 'set style line '//&
                        &trim(i2str(iplt))//' lc rgb '//&
```

```
                        &line_clrs(iplt)//' '//trim(line_style)
                    end do
                end if

                ! individual plots
                do iplt = 1,nplt
                    write(cmd_i,"(A)",iostat=istat) 'plot "'//trim(data_dir)//'/'//&
                        &trim(data_name_loc)//'.dat" using '//trim(i2str(iplt))//&
                        &':'//trim(i2str(nplt+iplt))//' title "'//&
                        &trim(var_names_loc(iplt))//'" '//&
                        &trim(loc_draw_op())
                end do

                ! finishing the command
                write(cmd_i,"(A)",iostat=istat) ''
            end subroutine draw_ex_animated_gnuplot

            !> \private Bokeh animated version: html output
            subroutine draw_ex_animated_bokeh
                ! initialize the script
                write(cmd_i,"(A)",iostat=istat) '# Note: it is necessary to first &
                    &run "bokeh serve" before calling python'
                write(cmd_i,"(A)",iostat=istat) ''
                write(cmd_i,"(A)",iostat=istat) 'from numpy import genfromtxt'
                write(cmd_i,"(A)",iostat=istat) 'from bokeh.plotting import &
                    &figure, output_file, save, show, curdoc'
                write(cmd_i,"(A)",iostat=istat) 'from bokeh.client import &
                    &push_session'
                write(cmd_i,"(A)",iostat=istat) 'from bokeh.driving import repeat'
                write(cmd_i,"(A)",iostat=istat) ''
                write(cmd_i,"(A)",iostat=istat) 'TOOLS="crosshair,pan,wheel_zoom,&
                    &box_zoom,reset,save"'
                write(cmd_i,"(A)",iostat=istat) ''
                write(cmd_i,"(A)",iostat=istat) 'output_file("'//trim(plot_dir)//&
                    &'/'//trim(draw_name)//'.html", title="'//trim(draw_name)//&
                    &'", mode="cdn") '
                write(cmd_i,"(A)",iostat=istat) ''
                write(cmd_i,"(A)",iostat=istat) 'data = genfromtxt("'//&
                    &trim(data_dir)//'/'//trim(data_name_loc)//'.dat")'
                write(cmd_i,"(A)",iostat=istat) ''
                write(cmd_i,"(A)",iostat=istat) 'p = figure(toolbar_location=&
                    &"above",'//'tools=TOOLS,active_drag="pan",active_scroll=&
                    &"wheel_zoom",'//'plot_width=600, plot_height=600)'
                write(cmd_i,"(A)",iostat=istat) ''

                ! write extra options
                if (present(extra_ops)) write(cmd_i,"(A)",iostat=istat) &
                    &trim(extra_ops)

                ! individual plot for first plot
                write(cmd_i,"(A)",iostat=istat) 'line = p.line(data[:,0],data[:,'//&
                    &trim(i2str(nplt))//'],'//trim(loc_draw_op(1))//',legend="'//&
                    &trim(var_names_loc(1))//'")'
                ! plot the circles
                write(cmd_i,"(A)",iostat=istat) 'circle = p.circle(data[:,'//&
                    &trim(i2str(0))//'],data[:,'//trim(i2str(nplt))//'],'//&
                    &trim(loc_draw_op(2))//')'

                ! finishing the command (needs to use show)
                write(cmd_i,"(A)",iostat=istat) ''
                write(cmd_i,"(A)",iostat=istat) 'session = push_session(curdoc())'
                write(cmd_i,"(A)",iostat=istat) 'session.show(p)'
                write(cmd_i,"(A)",iostat=istat) ''
                write(cmd_i,"(A)",iostat=istat) 'seq = range(0,'//&
                    &trim(i2str(nplt))//')'
                write(cmd_i,"(A)",iostat=istat) '@repeat(seq)'
                write(cmd_i,"(A)",iostat=istat) 'def update(i):'
                write(cmd_i,"(A)",iostat=istat) '    line.data_source.data["x"] = &
                    &data[:,i]'
                write(cmd_i,"(A)",iostat=istat) '    line.data_source.data["y"] = &
                    &data[:,'//trim(i2str(nplt))//'+i]'
                write(cmd_i,"(A)",iostat=istat) '    circle.data_source.data["x"] = &
                    &data[:,i]'
                write(cmd_i,"(A)",iostat=istat) '    circle.data_source.data["y"] = &
                    &data[:,'//&
                    &trim(i2str(nplt))//'+i]'
                write(cmd_i,"(A)",iostat=istat) ''
                write(cmd_i,"(A)",iostat=istat) 'curdoc().add_periodic_callback(&
                    &update, '//trim(i2str(delay_loc*10))//')'
                write(cmd_i,"(A)",iostat=istat) 'session.loop_until_closed()'
            end subroutine draw_ex_animated_bokeh

            !> \private gets ranges for animated plot
            subroutine get_ranges(ranges)
                ! input / output
                real(dp), intent(inout) :: ranges(:,:)                          ! x and y range, and z range (if 3D) of plot

```

```
            ! local variables
            integer :: data_i                                           ! file number of data file
            real(dp), allocatable :: loc_data(:)                        ! one line of data
            character(len=1) :: loc_data_char                           ! local first char


            ! open data file
            open(nextunit(data_i),file=data_dir//'/'//trim(data_name_loc)//&
                &'.dat',status='old',iostat=istat)

            if (istat.eq.0) then
                ! set up loc_data
                allocate(loc_data(2*nplt))

                ! read the data file
                istat = 0
                do while (istat.eq.0)
                    read(data_i,*,iostat=istat) loc_data_char           ! read first character of data
                    if (istat.eq.0) then                                ! read succesful
                        if (loc_data_char.ne.'#') then                  ! exclude comment lines
                            backspace(unit=data_i)                      ! go back one line
                            read(data_i,*,iostat=istat) loc_data        ! read data again, but now ful
                            ranges(1,1) = &
                                &min(ranges(1,1),minval(loc_data(1:nplt)))
                            ranges(1,2) = &
                                &max(ranges(1,2),maxval(loc_data(1:nplt)))
                            ranges(2,1) = &
                                &min(ranges(2,1),&
                                &minval(loc_data(nplt+1:2*nplt)))
                            ranges(2,2) = &
                                &max(ranges(2,2),&
                                &maxval(loc_data(nplt+1:2*nplt)))
                        end if
                    end if
                end do
                !write(*,*) 'ranges(1,:) = ', ranges(1,:)
                !write(*,*) 'ranges(2,:) = ', ranges(2,:)
                !write(*,*) 'ranges(3,:) = ', ranges(3,:)

                ! close data file
                close(data_i,iostat=istat)
            end if
        end subroutine get_ranges

        !> \private sets local draw options either from pre-defined line styles or through
        ! user specified option, for GNUPlot or Bokeh
        ! As for  Bokeh, there are  two default  draw options possible  (for the
        ! lines and for the points), there is an option to select between them.
        function loc_draw_op(Bokeh_style)
            ! input / output
            character(len=max_str_ln) :: loc_draw_op                    ! local drawing option
            integer, intent(in), optional :: bokeh_style                ! lines(1) or points (2)

            select case (ex_plot_style_loc)
                case (1)                                                ! GNUPlot
                    if (n_draw_ops.gt.0) then
                        loc_draw_op = trim(draw_ops(mod(iplt-1,n_draw_ops)+1))
                    else
                        loc_draw_op = 'with linespoints linestyle '//&
                            &trim(i2str(mod(iplt-1,size(line_clrs))+1))
                    end if
                case (2)                                                ! Bokeh / Mayavi
                    if (n_draw_ops.gt.0) then
                        loc_draw_op = trim(draw_ops(mod(iplt-1,n_draw_ops)+1))
                    else
                        select case (bokeh_style)
                            case (1)                                    ! lines
                                loc_draw_op = 'line_color='//&
                                    &line_clrs(mod(iplt-1,size(line_clrs))+1)//&
                                    &',line_width=2'
                            case (2)                                    ! points
                                loc_draw_op = 'color='//&
                                    &line_clrs(mod(iplt-1,size(line_clrs))+1)//&
                                    &',size=5'
                            case default
                                loc_draw_op = ''
                        end select
                    end if
            end select
        end function loc_draw_op
    end subroutine draw_ex

    !> Takes  two input  vectors and  plots these  as well  as the  relative and
    !! absolute difference in a HDF5 file.
    !!
    !! This is similar to a basic version of output_ops.plot_hdf5().
    !!
```

```fortran
    !! Optionally, an output message can be displayed on screen with the maximum
    !! relative and absolute error.
    subroutine plot_diff_hdf5(A,B,file_name,tot_dim,loc_offset,descr,&
        &output_message)

        ! input / output
        real(dp), intent(in) :: a(:,:,:)                                        !< vector A
        real(dp), intent(in) :: b(:,:,:)                                        !< vector B
        character(len=*), intent(in) :: file_name                               !< name of plot
        integer, intent(in), optional :: tot_dim(3)                             !< total dimensions of the arrays
        integer, intent(in), optional :: loc_offset(3)                          !< offset of local dimensions
        character(len=*), intent(in), optional :: descr                         !< description
        logical, intent(in), optional :: output_message                         !< whether to display a message or not

        ! local variables
        real(dp), allocatable :: plot_var(:,:,:,:)                              ! variable containing plot
        integer :: tot_dim_loc(4)                                               ! local version of tot_dim
        integer :: loc_offset_loc(4)                                            ! local version of loc_offset
        character(len=max_str_ln) :: var_names(5)                               ! names of variables in plot
        logical :: output_message_loc                                           ! local version of output_message
        real(dp) :: lim_lo                                                      ! lower limit of errors
        real(dp) :: lim_hi                                                      ! upper limit of errors
        real(dp) :: err_av                                                      ! average error
        real(dp), allocatable :: tot_lim(:)                                     ! total lower or upper limit
        real(dp), allocatable :: tot_err(:)                                     ! total sum of errors
        integer :: istat                                                        ! status
        logical :: ind_plot                                                     ! individual plot or not

        ! set up local tot_dim and loc_offset
        tot_dim_loc = [shape(a),5]
        if (present(tot_dim)) tot_dim_loc = [tot_dim,5]
        loc_offset_loc = 0
        if (present(loc_offset)) loc_offset_loc = [loc_offset,5]

        ! tests
        if (size(a,1).ne.size(b,1) .or. size(a,2).ne.size(b,2) .or. &
            &size(a,3).ne.size(b,3)) then
            call writo('in plot_diff_HDF5, A and B need to have the correct &
                &size',persistent=.true.,warning=.true.)
            return
        end if

        ! set up local ouput message
        output_message_loc = .false.
        if (present(output_message)) output_message_loc = output_message

        ind_plot = .false.
        if (tot_dim_loc(1).eq.size(a,1) .and. tot_dim_loc(2).eq.size(a,2) &
            &.and. tot_dim_loc(3).eq.size(a,3)) ind_plot = .true.

        ! set up plot_var
        allocate(plot_var(size(a,1),size(a,2),size(a,3),5))
        plot_var(:,:,:,1) = a
        plot_var(:,:,:,2) = b
        plot_var(:,:,:,3) = diff(a,b,shape(a),rel=.true.)                       ! rel. diff.
        plot_var(:,:,:,4) = log10(abs(diff(a,b,shape(a),rel=.true.)))           ! log of abs. value of rel. diff.
        plot_var(:,:,:,5) = diff(a,b,shape(a),rel=.false.)                      ! abs. diff.

        ! set up var_names
        var_names(1) = 'v1'
        var_names(2) = 'v2'
        var_names(3) = 'rel v1 - v2'
        var_names(4) = 'log abs rel v1 - v2'
        var_names(5) = 'abs v1 - v2'

        ! plot
        call plot_hdf5_arr(var_names,file_name,plot_var,tot_dim=tot_dim_loc,&
            &loc_offset=loc_offset_loc,col=1,descr=descr)

        ! output message if requested
        if (output_message_loc) then
            call writo('Information about errors:',persistent=.true.)
            call lvl_ud(1)
            ! relative error
            call stats(tot_dim_loc(1:3),plot_var(:,:,:,3),lim_lo,lim_hi,err_av)
            call writo(trim(r2strt(lim_lo))//' < rel. err. < '//&
                &trim(r2strt(lim_hi))//', average value: '//&
                &trim(r2strt(err_av)),persistent=.true.)
            ! log of absolute relative error
            call stats(tot_dim_loc(1:3),plot_var(:,:,:,4),lim_lo,lim_hi,err_av)
            call writo(trim(r2strt(lim_lo))//' < log(abs(rel. err.)) < '//&
                &trim(r2strt(lim_hi))//', average value: '//&
                &trim(r2strt(err_av)),persistent=.true.)
            ! absolute error
            call stats(tot_dim_loc(1:3),plot_var(:,:,:,5),lim_lo,lim_hi,err_av)
            call writo(trim(r2strt(lim_lo))//' < abs. err. < '//&
                &trim(r2strt(lim_hi))//', average value: '//&
```

```
                &trim(r2strt(err_av)),persistent=.true.)
            call lvl_ud(-1)
        end if
    contains
        ! returns relative or absolute difference between inputs A and B
        !> \private
        function diff(A,B,dims,rel) result(C)
            use num_vars, only: tol_zero

            ! input / output
            real(dp), intent(in) :: a(:,:,:)                                ! input A
            real(dp), intent(in) :: b(:,:,:)                                ! input B
            integer, intent(in) :: dims(3)                                  ! dimensions of A and B
            logical, intent(in) :: rel                                      ! .true. if relative and .false. if absolute error
            real(dp) :: c(dims(1),dims(2),dims(3))                          ! output C

            ! return output
            if (rel) then
                c = 2*(a-b)/max(tol_zero,(abs(a)+abs(b)))
            else
                c = abs(a-b)
            end if
        end function diff

        ! returns limits and average value
        !> \private
        subroutine stats(tot_dims,var,lim_lo,lim_hi,err_av)
            use mpi_utilities, only: get_ser_var

            ! input / output
            integer, intent(in) :: tot_dims(3)                              ! total dimensions of grid
            real(dp), intent(in) :: var(:,:,:)                              ! input variable for which to calculate statistics
            real(dp), intent(inout) :: lim_lo                               ! lower limit of errors
            real(dp), intent(inout) :: lim_hi                               ! upper limit of errors
            real(dp), intent(inout) :: err_av                               ! average error

            ! local variables
            real(dp) :: sum_err                                             ! sum of errors

            ! calculate limits on absolute error
            lim_lo = minval(var)
            lim_hi = maxval(var)
            sum_err = sum(var)

            ! get most stringent limits from all processes
            if (.not.ind_plot) then
                istat = get_ser_var([lim_lo],tot_lim)
                chckstt
                lim_lo = minval(tot_lim)
                istat = get_ser_var([lim_hi],tot_lim)
                chckstt
                lim_hi = maxval(tot_lim)
                istat = get_ser_var([sum_err],tot_err)
                chckstt
                sum_err = sum(tot_err)
            end if

            ! calculate average error
            err_av = sum_err/product(tot_dims)
        end subroutine
    end subroutine plot_diff_hdf5

    !> Executes command line, or displays a message if disabled.
    !!
    !! It also keeps a log of all shell commands executed.
    subroutine use_execute_command_line(command,exitstat,cmdstat,cmdmsg)
        use num_vars, only: do_execute_command_line, prog_name, &
            &shell_commands_name, shell_commands_i
#if ( lwith_intel && !lwith_gnu)
        use ifport
#endif

        ! input / output
        character(len=*), intent(in) :: command                             !< command to execute
        integer, intent(inout), optional :: exitstat                        !< exit status
        integer, intent(inout), optional :: cmdstat                         !< command status
        character(len=*), intent(inout), optional :: cmdmsg                 !< command message

        ! local variables
        integer :: istat                                                    ! status
        character(len=max_str_ln) :: full_name                              ! full name

        ! set up the full shell commands file name
        full_name = prog_name//'_'//trim(shell_commands_name)//'.sh'

```

```
        ! write the command to the file
        open(shell_commands_i,file=trim(full_name),status='old',&
            &position='append',iostat=istat)
        if (istat.eq.0) then
            write(shell_commands_i,'(A)',iostat=istat) trim(command)
            close(shell_commands_i,iostat=istat)
        else
            call writo('Failed to write to shell commands log file "'&
                &//trim(full_name)//'"',warning=.true.)
        end if

        ! initialize stati
        if (present(exitstat)) exitstat = 0
        if (present(cmdstat)) cmdstat = 0
        if (present(cmdmsg)) cmdmsg = ''

        ! execute command line
        if (do_execute_command_line) then
#if ( lwith_intel && !lwith_gnu)
            exitstat = system(command)
#else
            call execute_command_line(command,exitstat=exitstat,&
                &cmdstat=cmdstat,cmdmsg=cmdmsg)
#endif
        else
            call writo('Command added to log file "'//trim(full_name)//'":')
            call lvl_ud(1)
            call writo(command)
            call lvl_ud(-1)
            call writo('Run with "--do_execute_command_line" to execute &
                &command line instead')
        end if
    end subroutine
end module output_ops
```

D.64 /Users/toonweyens/Code/PB3D/Modules/PB3D_ops.f90 File Reference

Modules

- module pb3d_ops

  Operations on PB3D output.

Functions/Subroutines

- integer function, public pb3d_ops::reconstruct_pb3d_in (data_name)

  Reconstructs the input variables from PB3D HDF5 output.
- integer function, public pb3d_ops::reconstruct_pb3d_grid (grid, data_name, rich_lvl, tot_rich, lim_pos, grid_limits)

  Reconstructs grid variables from PB3D HDF5 output.
- integer function, public pb3d_ops::reconstruct_pb3d_eq_1 (grid_eq, eq, data_name, lim_pos)

  Reconstructs the equilibrium variables from PB3D HDF5 output.
- integer function, public pb3d_ops::reconstruct_pb3d_eq_2 (grid_eq, eq, data_name, rich_lvl, tot_rich, lim_pos)

  Reconstructs the equilibrium variables from PB3D HDF5 output.
- integer function, public pb3d_ops::reconstruct_pb3d_x_1 (mds, grid_x, x, data_name, rich_lvl, tot_rich, lim_sec_x, lim_pos)

  Reconstructs the vectorial perturbation variables from PB3D HDF5 output.
- integer function, public pb3d_ops::reconstruct_pb3d_x_2 (mds, grid_x, x, data_name, rich_lvl, tot_rich, lim_sec_x, lim_pos, is_field_averaged)

  Reconstructs the tensorial perturbation variables from PB3D HDF5 output.
- integer function, public pb3d_ops::reconstruct_pb3d_vac (vac, data_name, rich_lvl)

Reconstructs the vacuum variables from PB3D HDF5 output.

- integer function, public pb3d_ops::reconstruct_pb3d_sol (mds, grid_sol, sol, data_name, rich_lvl, lim↩_sec_sol, lim_pos)

  Reconstructs the solution variables from PB3D HDF5 output.

- integer function, public pb3d_ops::get_pb3d_grid_size (n, grid_name, rich_lvl, tot_rich)

  get grid size

D.65 PB3D_ops.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Operations on PB3D output.
!!
!! \note If  you have parallel jobs,  if the reconstruction routines  are called
!! for  multiple  equilibrium  jobs,  this  can only  be  done  after  the  last
!! equilibrium job is finished. There are no checks for this.
!-----------------------------------------------------------------------------!
module pb3d_ops
#include <PB3D_macros.h>
    use str_utilities
    use messages
    use output_ops
    use num_vars, only: dp, pi, max_str_ln, max_name_ln, iu, min_pb3d_version
    use grid_vars, only: grid_type
    use eq_vars, only: eq_1_type, eq_2_type
    use x_vars, only: x_1_type, x_2_type, modes_type
    use vac_vars, only: vac_type
    use sol_vars, only: sol_type
    use hdf5_vars, only: dealloc_var_1d, var_1d_type
    use pb3d_utilities, only: setup_rich_id, setup_par_id

    implicit none
    private
    public reconstruct_pb3d_in, reconstruct_pb3d_grid, reconstruct_pb3d_eq_1, &
        &reconstruct_pb3d_eq_2, reconstruct_pb3d_x_1, reconstruct_pb3d_x_2, &
        &reconstruct_pb3d_vac, reconstruct_pb3d_sol, get_pb3d_grid_size

    ! global variables
    real(dp), allocatable :: dum_1D(:)                                      ! dummy variables
    real(dp), allocatable :: dum_2D(:,:)                                    ! dummy variables
    real(dp), allocatable :: dum_3D(:,:,:)                                  ! dummy variables
    real(dp), allocatable :: dum_4D(:,:,:,:)                                ! dummy variables
    real(dp), allocatable :: dum_6D(:,:,:,:,:,:)                            ! dummy variables
    real(dp), allocatable :: dum_7D(:,:,:,:,:,:,:)                          ! dummy variables

contains
    !> Reconstructs the input variables from PB3D HDF5 output.
    !!
    !! \return ierr
    integer function reconstruct_pb3d_in(data_name) result(ierr)
        use num_vars, only: eq_style, rho_style, use_pol_flux_e, max_deriv, &
            &use_pol_flux_f, use_normalization, norm_disc_prec_eq, pb3d_name, &
            &norm_disc_prec_x, norm_style, u_style, x_style, prog_style, &
            &matrix_slepc_style, bc_style, ev_style, norm_disc_prec_sol, &
            &norm_disc_style_sol, ev_bc, magn_int_style, k_style, &
            &alpha_style, x_grid_style, debug_version, max_njq_change
        use hdf5_ops, only: read_hdf5_arr
        use pb3d_utilities, only: conv_1d2nd
        use eq_vars, only: r_0, pres_0, b_0, psi_0, rho_0, t_0, vac_perm, &
            &max_flux_e, max_flux_f
        use grid_vars, onLy: n_r_in, n_r_eq, n_r_sol, min_alpha, max_alpha, &
            &n_alpha
        use x_vars, only: min_r_sol, max_r_sol, min_sec_x, max_sec_x, prim_x, &
            &n_mod_x
        use helena_vars, only: chi_h, flux_p_h, flux_t_h, r_h, z_h, nchi, ias, &
            &q_saf_h, rot_t_h, pres_h, rbphi_h
        use vmec_vars, only: is_freeb_v, mnmax_v, mpol_v, ntor_v, is_asym_v, &
            &gam_v, r_v_c, r_v_s, z_v_c, z_v_s, l_v_c, l_v_s, jac_v_c, &
            &jac_v_s, mn_v, rot_t_v, q_saf_v, pres_v, flux_t_v, flux_p_v, &
            &nfp_v, b_v_sub_c, b_v_sub_s
        use helena_vars, only: h_h_11, h_h_12, h_h_33
#if ldebug
        use vmec_vars, only: b_v_c, b_v_s, j_v_sup_int
#endif

        character(*), parameter :: rout_name = 'reconstruct_PB3D_in'
```

```

        ! input / output
        character(len=*), intent(in) :: data_name                               !< name of input variables

        ! local variables
        character(len=max_str_ln) :: err_msg                                    ! error message
        type(var_1d_type) :: var_1d                                             ! 1D variable
        real(dp), parameter :: tol_version = 1.e-4_dp                           ! tolerance for version control
        real(dp) :: pb3d_version                                                ! version of PB3D variable read
        logical :: debug_version_pb3d                                           ! debug version of in
        character(len=max_str_ln) :: use_debug_str(2)                           ! using debug or not

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Reconstructing input variables from PB3D output')
        call lvl_ud(1)

        ! misc_in
        ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'misc_in')
        chckerr('')
        call conv_1d2nd(var_1d,dum_1d)
        pb3d_version = dum_1d(1)
        eq_style = nint(dum_1d(2))
        rho_style = nint(dum_1d(3))
        r_0 = dum_1d(4)
        pres_0 = dum_1d(5)
        b_0 = dum_1d(6)
        psi_0 = dum_1d(7)
        rho_0 = dum_1d(8)
        t_0 = dum_1d(9)
        vac_perm = dum_1d(10)
        use_pol_flux_e = .false.
        use_pol_flux_f = .false.
        use_normalization = .false.
        if (dum_1d(11).gt.0) use_pol_flux_e = .true.
        if (dum_1d(12).gt.0) use_pol_flux_f = .true.
        if (dum_1d(13).gt.0) use_normalization = .true.
        norm_disc_prec_eq = nint(dum_1d(14))
        n_r_in = nint(dum_1d(15))
        n_r_eq = nint(dum_1d(16))
        n_r_sol = nint(dum_1d(17))
        max_flux_e = dum_1d(18)
        max_flux_f = dum_1d(19)
        debug_version_pb3d = .false.
        if (dum_1d(20).gt.0) debug_version_pb3d = .true.
        call dealloc_var_1d(var_1d)

        ! tests
        select case (prog_style)
            case (1)                                                            ! PB3D
                ! do nothing
            case (2)                                                            ! POST
                call writo('Run tests')
                call lvl_ud(1)

                call writo('PB3D version '//trim(r2strt(pb3d_version)))
                if (pb3d_version.lt.min_pb3d_version*(1-tol_version)) then
                    ierr = 1
                    err_msg = 'Need at least PB3D version '//&
                        &trim(r2strt(min_pb3d_version))
                    chckerr(err_msg)
                end if

                if (debug_version_pb3d) call writo('debug version')

                if (debug_version_pb3d.neqv.debug_version) then
                    ierr = 1
                    if (debug_version_pb3d) then
                        use_debug_str(1) = 'uses debug version'
                    else
                        use_debug_str(1) = 'uses release version'
                    end if
                    if (debug_version) then
                        use_debug_str(2) = 'uses debug version'
                    else
                        use_debug_str(2) = 'uses release version'
                    end if
                    call writo('The PB3D output '//trim(use_debug_str(1))//&
                        &' but POST '//trim(use_debug_str(2)))
                    err_msg = 'Need to use debug version for both, or not for &
                        &both'
                    chckerr(err_msg)
                end if

                call lvl_ud(-1)
```

```
        end select

        ! variables depending on equilibrium style
        select case (eq_style)
            case (1)                                                        ! VMEC
                ! misc_in_V
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                    &'misc_in_V')
                chckerr('')
                call conv_1d2nd(var_1d,dum_1d)
                is_asym_v = .false.
                is_freeb_v = .false.
                if (dum_1d(1).gt.0) is_asym_v = .true.
                if (dum_1d(2).gt.0) is_freeb_v = .true.
                mnmax_v = nint(dum_1d(3))
                mpol_v = nint(dum_1d(4))
                ntor_v = nint(dum_1d(5))
                nfp_v = nint(dum_1d(6))
                gam_v = dum_1d(7)
                call dealloc_var_1d(var_1d)

                ! flux_t_V
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                    &'flux_t_V')
                chckerr('')
                call conv_1d2nd(var_1d,dum_2d)
                allocate(flux_t_v(n_r_eq,0:max_deriv+1))
                flux_t_v = dum_2d
                call dealloc_var_1d(var_1d)

                ! flux_p_V
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                    &'flux_p_V')
                chckerr('')
                call conv_1d2nd(var_1d,dum_2d)
                allocate(flux_p_v(n_r_eq,0:max_deriv+1))
                flux_p_v = dum_2d
                call dealloc_var_1d(var_1d)

                ! pres_V
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'pres_V')
                chckerr('')
                call conv_1d2nd(var_1d,dum_2d)
                allocate(pres_v(n_r_eq,0:max_deriv+1))
                pres_v = dum_2d
                call dealloc_var_1d(var_1d)

                ! rot_t_V
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'rot_t_V')
                chckerr('')
                call conv_1d2nd(var_1d,dum_2d)
                allocate(rot_t_v(n_r_eq,0:max_deriv+1))
                rot_t_v = dum_2d
                call dealloc_var_1d(var_1d)

                ! q_saf_V
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'q_saf_V')
                chckerr('')
                call conv_1d2nd(var_1d,dum_2d)
                allocate(q_saf_v(n_r_eq,0:max_deriv+1))
                q_saf_v = dum_2d
                call dealloc_var_1d(var_1d)

                ! mn_V
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'mn_V')
                chckerr('')
                call conv_1d2nd(var_1d,dum_2d)
                allocate(mn_v(mnmax_v,2))
                mn_v = nint(dum_2d)
                call dealloc_var_1d(var_1d)

                ! RZL_V
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'RZL_V')
                chckerr('')
                call conv_1d2nd(var_1d,dum_4d)
                allocate(r_v_c(mnmax_v,n_r_eq,0:max_deriv+1))
                allocate(r_v_s(mnmax_v,n_r_eq,0:max_deriv+1))
                allocate(z_v_c(mnmax_v,n_r_eq,0:max_deriv+1))
                allocate(z_v_s(mnmax_v,n_r_eq,0:max_deriv+1))
                allocate(l_v_c(mnmax_v,n_r_eq,0:max_deriv+1))
                allocate(l_v_s(mnmax_v,n_r_eq,0:max_deriv+1))
                r_v_c = dum_4d(:,:,:,1)
                r_v_s = dum_4d(:,:,:,2)
                z_v_c = dum_4d(:,:,:,3)
                z_v_s = dum_4d(:,:,:,4)
                l_v_c = dum_4d(:,:,:,5)
                l_v_s = dum_4d(:,:,:,6)
```

```
                call dealloc_var_1d(var_1d)

                ! jac_V
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'jac_V')
                chckerr('')
                call conv_1d2nd(var_1d,dum_4d)
                allocate(jac_v_c(mnmax_v,n_r_eq,0:max_deriv))
                allocate(jac_v_s(mnmax_v,n_r_eq,0:max_deriv))
                jac_v_c = dum_4d(:,:,:,1)
                jac_v_s = dum_4d(:,:,:,2)
                call dealloc_var_1d(var_1d)

                ! B_V_sub
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'B_V_sub')
                chckerr('')
                call conv_1d2nd(var_1d,dum_4d)
                allocate(b_v_sub_c(mnmax_v,n_r_eq,3))
                allocate(b_v_sub_s(mnmax_v,n_r_eq,3))
                b_v_sub_c = dum_4d(:,:,:,1)
                b_v_sub_s = dum_4d(:,:,:,2)
                call dealloc_var_1d(var_1d)

#if ldebug
                ! B_V
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'B_V')
                chckerr('')
                call conv_1d2nd(var_1d,dum_3d)
                allocate(b_v_c(mnmax_v,n_r_eq))
                allocate(b_v_s(mnmax_v,n_r_eq))
                b_v_c = dum_3d(:,:,1)
                b_v_s = dum_3d(:,:,2)
                call dealloc_var_1d(var_1d)

                ! J_V_sup_int
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                    &'J_V_sup_int')
                chckerr('')
                call conv_1d2nd(var_1d,dum_2d)
                allocate(j_v_sup_int(n_r_eq,1:2))
                j_v_sup_int = dum_2d
                call dealloc_var_1d(var_1d)
#endif
            case (2)                                                    ! HELENA
                ! misc_in_H
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                    &'misc_in_H')
                chckerr('')
                call conv_1d2nd(var_1d,dum_1d)
                ias = nint(dum_1d(1))
                nchi= nint(dum_1d(2))
                call dealloc_var_1d(var_1d)

                ! RZ_H
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'RZ_H')
                chckerr('')
                call conv_1d2nd(var_1d,dum_3d)
                allocate(r_h(nchi,n_r_eq))
                allocate(z_h(nchi,n_r_eq))
                r_h = dum_3d(:,:,1)
                z_h = dum_3d(:,:,2)
                call dealloc_var_1d(var_1d)

                ! chi_H
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'chi_H')
                chckerr('')
                call conv_1d2nd(var_1d,dum_1d)
                allocate(chi_h(nchi))
                chi_h = dum_1d
                call dealloc_var_1d(var_1d)

                ! flux_p_H
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                    &'flux_p_H')
                chckerr('')
                call conv_1d2nd(var_1d,dum_2d)
                allocate(flux_p_h(n_r_eq,0:max_deriv+1))
                flux_p_h = dum_2d
                call dealloc_var_1d(var_1d)

                ! flux_t_H
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                    &'flux_t_H')
                chckerr('')
                call conv_1d2nd(var_1d,dum_2d)
                allocate(flux_t_h(n_r_eq,0:max_deriv+1))
                flux_t_h = dum_2d
                call dealloc_var_1d(var_1d)
```

```
                ! q_saf_H
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'q_saf_H')
                chckerr('')
                call conv_1d2nd(var_1d,dum_2d)
                allocate(q_saf_h(n_r_eq,0:max_deriv+1))
                q_saf_h = dum_2d
                call dealloc_var_1d(var_1d)

                ! rot_t_H
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'rot_t_H')
                chckerr('')
                call conv_1d2nd(var_1d,dum_2d)
                allocate(rot_t_h(n_r_eq,0:max_deriv+1))
                rot_t_h = dum_2d
                call dealloc_var_1d(var_1d)

                ! pres_H
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'pres_H')
                chckerr('')
                call conv_1d2nd(var_1d,dum_2d)
                allocate(pres_h(n_r_eq,0:max_deriv+1))
                pres_h = dum_2d
                call dealloc_var_1d(var_1d)

                ! RBphi_H
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'RBphi_H')
                chckerr('')
                call conv_1d2nd(var_1d,dum_2d)
                allocate(rbphi_h(n_r_eq,0:max_deriv+1))
                rbphi_h = dum_2d
                call dealloc_var_1d(var_1d)

                ! h_H
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'h_H')
                chckerr('')
                call conv_1d2nd(var_1d,dum_3d)
                allocate(h_h_11(nchi,n_r_eq))
                allocate(h_h_12(nchi,n_r_eq))
                allocate(h_h_33(nchi,n_r_eq))
                h_h_11 = dum_3d(:,:,1)
                h_h_12 = dum_3d(:,:,2)
                h_h_33 = dum_3d(:,:,3)
                call dealloc_var_1d(var_1d)
        end select

        ! misc_X
        ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'misc_X')
        chckerr('')
        call conv_1d2nd(var_1d,dum_1d)
        prim_x = nint(dum_1d(1))
        n_mod_x = nint(dum_1d(2))
        min_sec_x = nint(dum_1d(3))
        max_sec_x = nint(dum_1d(4))
        norm_disc_prec_x = nint(dum_1d(5))
        norm_style = nint(dum_1d(6))
        u_style = nint(dum_1d(7))
        x_style = nint(dum_1d(8))
        matrix_slepc_style = nint(dum_1d(9))
        select case(prog_style)
            case (1)                                                    ! PB3D
                magn_int_style = nint(dum_1d(10))
            case (2)                                                    ! POST
                magn_int_style = 1                                      ! integration is done in volume, currently
only with trapezoidal rule
        end select
        k_style = nint(dum_1d(11))
        alpha_style = nint(dum_1d(12))
        x_grid_style = nint(dum_1d(13))
        min_alpha = dum_1d(14)
        max_alpha = dum_1d(15)
        n_alpha = nint(dum_1d(16))
        max_njq_change = dum_1d(17)
        call dealloc_var_1d(var_1d)

        ! misc_sol
        ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'misc_sol')
        chckerr('')
        call conv_1d2nd(var_1d,dum_1d)
        min_r_sol = dum_1d(1)
        max_r_sol = dum_1d(2)
        norm_disc_prec_sol = nint(dum_1d(3))
        norm_disc_style_sol = nint(dum_1d(4))
        bc_style(1) = nint(dum_1d(5))
        bc_style(2) = nint(dum_1d(6))
        ev_style = nint(dum_1d(7))
        ev_bc = dum_1d(8)
```

```
        call dealloc_var_1d(var_1d)

        ! user output
        call lvl_ud(-1)
        call writo('Input variables from PB3D output reconstructed')
    end function reconstruct_pb3d_in

    !> Reconstructs grid variables from PB3D HDF5 output.
    !!
    !! Also, if \c  rich_lvl is provided, <tt>_R_[rich_lvl]</tt>  is appended to
    !! the data name if it is > 0.
    !!
    !! With \c tot_rich  the information from previous Richardson  levels can be
    !! combined.
    !!
    !! Furthermore, using  \c lim_pos, you  can obtain a  subset of the  data by
    !! directly passing its  limits to the underlying HDF5  routine. This refers
    !! to the  position dimensions  only. If  provided, the  normal limits  of a
    !! divided grid refer to the subset, as in copy_grid().
    !!
    !! \note
    !!  -# <tt>grid_</tt> is added in front the data_name.
    !!  -# By  providing \c lim_pos  equal to 0  in the angular  dimensions, the
    !!  angular part can be discarded when reconstructing the grid.
    !!
    !! \return ierr
    integer function reconstruct_pb3d_grid(grid,data_name,rich_lvl,tot_rich,&
        &lim_pos,grid_limits) result(ierr)

        use num_vars, only: pb3d_name
        use hdf5_ops, only: read_hdf5_arr
        use pb3d_utilities, only: conv_1d2nd

        character(*), parameter :: rout_name = 'reconstruct_PB3D_grid'

        ! input / output
        type(grid_type), intent(inout) :: grid                                  !< grid
        character(len=*), intent(in) :: data_name                               !< name of grid
        integer, intent(in), optional :: rich_lvl                               !< Richardson level to reconstruct
        logical, intent(in), optional :: tot_rich                               !< whether to combine with previous Richardson levels
        integer, intent(in), optional :: lim_pos(3,2)                           !< position limits of subset of data
        integer, intent(in), optional :: grid_limits(2)                         !< i_limit of grid

        ! local variables
        type(var_1d_type) :: var_1d                                             ! 1D variable
        integer :: lim_mem(3,2)                                                 ! memory limits for variables
        integer :: n(3)                                                         ! total n
        integer :: id                                                           ! counter
        integer :: rich_lvl_loc                                                 ! local rich_lvl
        integer :: par_id(3)                                                    ! parallel indices (start, end, stride)
        integer :: par_id_mem(2)                                                ! parallel indices (start, end) in memory (stride 1)
        integer :: par_lim(2)                                                   ! parallel limits
        integer :: rich_id(2)                                                   ! richardson level indices (start, end)

        ! initialize ierr
        ierr = 0

        ! set up local rich_lvl
        rich_lvl_loc = 0
        if (present(rich_lvl)) rich_lvl_loc = rich_lvl

        ! setup rich_id
        rich_id = setup_rich_id(rich_lvl_loc,tot_rich)

        ! get total grid size
        ierr = get_pb3d_grid_size(n,data_name,rich_lvl,tot_rich)
        chckerr('')

        ! possibly change n to user-specified
        if (present(lim_pos)) then
            if (lim_pos(1,1).ge.0 .and. lim_pos(1,2).ge.0) n(1) = &
                &lim_pos(1,2)-lim_pos(1,1)+1
            if (lim_pos(2,1).ge.0 .and. lim_pos(2,2).ge.0) n(2) = &
                &lim_pos(2,2)-lim_pos(2,1)+1
            if (lim_pos(3,1).ge.0 .and. lim_pos(3,2).ge.0) n(3) = &
                &lim_pos(3,2)-lim_pos(3,1)+1
            where (lim_pos(:,1).eq.lim_pos(:,2) .and. lim_pos(:,1).eq.[0,0,0]) &
                &n = 0
        end if

        ! set up parallel limits
        par_lim = [1,n(1)]
        if (present(lim_pos)) then
            where (lim_pos(1,:).ge.0) par_lim = lim_pos(1,:)
        end if
```

```fortran

        ! create grid
        ierr = grid%init(n,i_lim=grid_limits)
        chckerr('')

        ! restore looping over richardson levels
        do id = rich_id(2),rich_id(1),-1
            ! setup par_id
            par_id = setup_par_id(grid,rich_lvl_loc,id,tot_rich=tot_rich,&
                &par_lim=par_lim,par_id_mem=par_id_mem)

            ! set up local limits for HDF5 reconstruction of full vars
            lim_mem(1,:) = par_id_mem
            lim_mem(2,:) = [1,grid%n(2)]
            lim_mem(3,:) = [1,grid%n(3)]
            if (present(lim_pos)) then
                lim_mem(2,:) = lim_pos(2,:)
                lim_mem(3,:) = lim_pos(3,:)
            end if

            ! r_F
            ierr = read_hdf5_arr(var_1d,pb3d_name,'grid_'//&
                &trim(data_name),'r_F',rich_lvl=id,lim_loc=lim_mem(3:3,:))
            chckerr('')
            call conv_1d2nd(var_1d,dum_1d)
            grid%r_F = dum_1d
            call dealloc_var_1d(var_1d)

            ! r_E
            ierr = read_hdf5_arr(var_1d,pb3d_name,'grid_'//&
                &trim(data_name),'r_E',rich_lvl=id,lim_loc=lim_mem(3:3,:))
            chckerr('')
            call conv_1d2nd(var_1d,dum_1d)
            grid%r_E = dum_1d
            call dealloc_var_1d(var_1d)

            ! loc_r_F
            grid%loc_r_F = grid%r_F(grid%i_min:grid%i_max)

            ! loc_r_E
            grid%loc_r_E = grid%r_E(grid%i_min:grid%i_max)

            ! overwrite local limits for HDF5 reconstruction of divided vars
            lim_mem(1,:) = par_id_mem
            lim_mem(2,:) = [1,grid%n(2)]
            lim_mem(3,:) = [grid%i_min,grid%i_max]
            if (present(lim_pos)) then
                lim_mem(2,:) = lim_pos(2,:)
                lim_mem(3,:) = lim_mem(3,:) + lim_pos(3,1) - 1                  ! take into account the grid limits (relative to position subset)
            end if

            ! only for 3D grids
            if (product(grid%n(1:2)).ne.0) then
                ! theta_F
                ierr = read_hdf5_arr(var_1d,pb3d_name,'grid_'//&
                    &trim(data_name),'theta_F',rich_lvl=id,lim_loc=lim_mem)
                chckerr('')
                call conv_1d2nd(var_1d,dum_3d)
                grid%theta_F(par_id(1):par_id(2):par_id(3),:,:) = dum_3d
                call dealloc_var_1d(var_1d)

                ! theta_E
                ierr = read_hdf5_arr(var_1d,pb3d_name,'grid_'//&
                    &trim(data_name),'theta_E',rich_lvl=id,lim_loc=lim_mem)
                chckerr('')
                call conv_1d2nd(var_1d,dum_3d)
                grid%theta_E(par_id(1):par_id(2):par_id(3),:,:) = dum_3d
                call dealloc_var_1d(var_1d)

                ! zeta_F
                ierr = read_hdf5_arr(var_1d,pb3d_name,'grid_'//&
                    &trim(data_name),'zeta_F',rich_lvl=id,lim_loc=lim_mem)
                chckerr('')
                call conv_1d2nd(var_1d,dum_3d)
                grid%zeta_F(par_id(1):par_id(2):par_id(3),:,:) = dum_3d
                call dealloc_var_1d(var_1d)

                ! zeta_E
                ierr = read_hdf5_arr(var_1d,pb3d_name,'grid_'//&
                    &trim(data_name),'zeta_E',rich_lvl=id,lim_loc=lim_mem)
                chckerr('')
                call conv_1d2nd(var_1d,dum_3d)
                grid%zeta_E(par_id(1):par_id(2):par_id(3),:,:) = dum_3d
                call dealloc_var_1d(var_1d)
            end if
        end do
```

```
    end function reconstruct_pb3d_grid

    !> Reconstructs the equilibrium variables from PB3D HDF5 output.
    !!
    !! Furthermore, using  \c lim_pos, you  can obtain a  subset of the  data by
    !! directly passing its  limits to the underlying HDF5  routine. This refers
    !! to the  position dimensions  only. If  provided, the  normal limits  of a
    !! divided grid refer to the subset, as in copy_grid().
    !!
    !! \return ierr
    integer function reconstruct_pb3d_eq_1(grid_eq,eq,data_name,lim_pos) &
        &result(ierr)                                                   ! flux version
        use num_vars, only: pb3d_name
        use hdf5_ops, only: read_hdf5_arr
        use pb3d_utilities, only: conv_1d2nd

        character(*), parameter :: rout_name = 'reconstruct_PB3D_eq_1'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                          !< equilibrium grid
        type(eq_1_type), intent(inout), optional :: eq                  !< flux equilibrium
        character(len=*), intent(in) :: data_name                       !< name to reconstruct
        integer, intent(in), optional :: lim_pos(1,2)                   !< position limits of subset of data

        ! local variables
        type(var_1d_type) :: var_1d                                     ! 1D variable
        integer :: lim_mem(2,2)                                         ! memory limits for variables

        ! initialize ierr
        ierr = 0

        ! prepare

        ! create equilibrium
        call eq%init(grid_eq,setup_e=.false.,setup_f=.true.)

        ! set up local limits for HDF5 reconstruction
        lim_mem(1,:) = [grid_eq%i_min,grid_eq%i_max]
        lim_mem(2,:) = [0,-1]                                           ! all derivatives, starting from 0
        if (present(lim_pos)) then
            lim_mem(1,:) = lim_mem(1,:) + lim_pos(1,1) - 1              ! take into account the grid limits (relative to position subset)
        end if

        ! restore variables

        ! pres_FD
        ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'pres_FD',&
            &lim_loc=lim_mem)
        chckerr('')
        call conv_1d2nd(var_1d,dum_2d)
        eq%pres_FD = dum_2d
        call dealloc_var_1d(var_1d)

        ! q_saf_FD
        ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'q_saf_FD',&
            &lim_loc=lim_mem)
        chckerr('')
        call conv_1d2nd(var_1d,dum_2d)
        eq%q_saf_FD = dum_2d
        call dealloc_var_1d(var_1d)

        ! rot_t_FD
        ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'rot_t_FD',&
            &lim_loc=lim_mem)
        chckerr('')
        call conv_1d2nd(var_1d,dum_2d)
        eq%rot_t_FD = dum_2d
        call dealloc_var_1d(var_1d)

        ! flux_p_FD
        ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'flux_p_FD',&
            &lim_loc=lim_mem)
        chckerr('')
        call conv_1d2nd(var_1d,dum_2d)
        eq%flux_p_FD = dum_2d
        call dealloc_var_1d(var_1d)

        ! flux_t_FD
        ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'flux_t_FD',&
            &lim_loc=lim_mem)
        chckerr('')
        call conv_1d2nd(var_1d,dum_2d)
        eq%flux_t_FD = dum_2d
        call dealloc_var_1d(var_1d)

        ! rho
```

```
        ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'rho',&
            &lim_loc=lim_mem(1:1,:))
        chckerr('')
        call conv_1d2nd(var_1d,dum_1d)
        eq%rho = dum_1d
        call dealloc_var_1d(var_1d)
    end function reconstruct_pb3d_eq_1

    !> Reconstructs the equilibrium variables from PB3D HDF5 output.
    !!
    !! Also, if \c  rich_lvl is provided, <tt>_R_[rich_lvl]</tt>  is appended to
    !! the data name if it is > 0.
    !!
    !! With \c tot_rich  the information from previous Richardson  levels can be
    !! combined.
    !!
    !! Furthermore, using  \c lim_pos, you  can obtain a  subset of the  data by
    !! directly passing its  limits to the underlying HDF5  routine. This refers
    !! to the  position dimensions  only. If  provided, the  normal limits  of a
    !! divided grid refer to the subset, as in copy_grid().
    !!
    !! \return ierr
    integer function reconstruct_pb3d_eq_2(grid_eq,eq,data_name,rich_lvl,&
        &tot_rich,lim_pos) result(ierr)                                    ! metric version
        use num_vars, only: pb3d_name
        use hdf5_ops, only: read_hdf5_arr
        use pb3d_utilities, only: conv_1d2nd

        character(*), parameter :: rout_name = 'reconstruct_PB3D_eq_2'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                             !< equilibrium grid
        type(eq_2_type), intent(inout) :: eq                               !< metric equilibrium
        character(len=*), intent(in) :: data_name                          !< name to reconstruct
        integer, intent(in), optional :: rich_lvl                          !< Richardson level to reconstruct
        logical, intent(in), optional :: tot_rich                          !< whether to combine with previous Richardson levels
        integer, intent(in), optional :: lim_pos(3,2)                      !< position limits of subset of data

        ! local variables
        type(var_1d_type) :: var_1d                                        ! 1D variable
        integer :: lim_mem(7,2)                                            ! memory limits for variables
        integer :: id                                                      ! counter
        integer :: rich_lvl_loc                                            ! local rich_lvl
        integer :: par_id(3)                                               ! parallel indices (start, end, stride)
        integer :: par_id_mem(2)                                           ! parallel indices (start, end) in memory (stride 1)
        integer :: par_lim(2)                                              ! parallel limits
        integer :: rich_id(2)                                              ! richardson level indices (start, end)

        ! initialize ierr
        ierr = 0

        ! set up local rich_lvl
        rich_lvl_loc = 0
        if (present(rich_lvl)) rich_lvl_loc = rich_lvl

        ! setup rich_id
        rich_id = setup_rich_id(rich_lvl_loc,tot_rich)

        ! set up parallel limits
        par_lim = [1,grid_eq%n(1)]
        if (present(lim_pos)) then
            where (lim_pos(1,:).ge.0) par_lim = lim_pos(1,:)
        end if

        ! create equilibrium
        call eq%init(grid_eq,setup_e=.false.,setup_f=.true.)

        ! restore looping over richardson levels
        do id = rich_id(2),rich_id(1),-1
            ! setup par_id
            par_id = setup_par_id(grid_eq,rich_lvl_loc,id,tot_rich=tot_rich,&
                &par_lim=par_lim,par_id_mem=par_id_mem)

            ! set up local limits for HDF5 reconstruction
            lim_mem(1,:) = par_id_mem
            lim_mem(2,:) = [1,grid_eq%n(2)]
            lim_mem(3,:) = [grid_eq%i_min,grid_eq%i_max]
            lim_mem(4,:) = [-1,-1]
            lim_mem(5,:) = [-1,-1]
            lim_mem(6,:) = [-1,-1]
            lim_mem(7,:) = [-1,-1]
            if (present(lim_pos)) then
                lim_mem(2,:) = lim_pos(2,:)
                lim_mem(3,:) = lim_mem(3,:) + lim_pos(3,1) - 1             ! take into account the grid limits (relative to position subset)
```

```
            end if

            ! g_FD
            ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'g_FD',&
                &rich_lvl=id,lim_loc=lim_mem)
            chckerr('')
            call conv_1d2nd(var_1d,dum_7d)
            eq%g_FD(par_id(1):par_id(2):par_id(3),:,:,:,:,:,:) = dum_7d
            call dealloc_var_1d(var_1d)

            ! h_FD
            ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'h_FD',&
                &rich_lvl=id,lim_loc=lim_mem)
            chckerr('')
            call conv_1d2nd(var_1d,dum_7d)
            eq%h_FD(par_id(1):par_id(2):par_id(3),:,:,:,:,:,:) = dum_7d
            call dealloc_var_1d(var_1d)

            ! jac_FD
            ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'jac_FD',&
                &rich_lvl=id,lim_loc=lim_mem(1:6,:))
            chckerr('')
            call conv_1d2nd(var_1d,dum_6d)
            eq%jac_FD(par_id(1):par_id(2):par_id(3),:,:,:,:,:) = dum_6d
            call dealloc_var_1d(var_1d)

            ! S
            ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'S',&
                &rich_lvl=id,lim_loc=lim_mem(1:3,:))
            chckerr('')
            call conv_1d2nd(var_1d,dum_3d)
            eq%S(par_id(1):par_id(2):par_id(3),:,:) = dum_3d
            call dealloc_var_1d(var_1d)

            ! kappa_n
            ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                &'kappa_n',rich_lvl=id,lim_loc=lim_mem(1:3,:))
            chckerr('')
            call conv_1d2nd(var_1d,dum_3d)
            eq%kappa_n(par_id(1):par_id(2):par_id(3),:,:) = dum_3d
            call dealloc_var_1d(var_1d)

            ! kappa_g
            ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                &'kappa_g',rich_lvl=id,lim_loc=lim_mem(1:3,:))
            chckerr('')
            call conv_1d2nd(var_1d,dum_3d)
            eq%kappa_g(par_id(1):par_id(2):par_id(3),:,:) = dum_3d
            call dealloc_var_1d(var_1d)

            ! sigma
            ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'sigma',&
                &rich_lvl=id,lim_loc=lim_mem(1:3,:))
            chckerr('')
            call conv_1d2nd(var_1d,dum_3d)
            eq%sigma(par_id(1):par_id(2):par_id(3),:,:) = dum_3d
            call dealloc_var_1d(var_1d)
        end do
    end function reconstruct_pb3d_eq_2

    !> Reconstructs the vectorial perturbation variables from PB3D HDF5 output.
    !!
    !! Also, if \c  rich_lvl is provided, <tt>_R_[rich_lvl]</tt>  is appended to
    !! the data name if it is > 0.
    !!
    !! With \c tot_rich  the information from previous Richardson  levels can be
    !! combined.
    !!
    !! Furthermore, using  \c lim_pos, you  can obtain a  subset of the  data by
    !! directly passing its  limits to the underlying HDF5  routine. This refers
    !! to the  position dimensions  only. If  provided, the  normal limits  of a
    !! divided grid refer to the subset, as in copy_grid().
    !!
    !! \c lim_sec_X, on the other hand, selects a range of mode numbers.
    !!
    !! \return ierr
    integer function reconstruct_pb3d_x_1(mds,grid_X,X,data_name,rich_lvl,&
        &tot_rich,lim_sec_X,lim_pos) result(ierr)
        use num_vars, only: pb3d_name
        use x_vars, only: n_mod_x
        use hdf5_ops, only: read_hdf5_arr
        use pb3d_utilities, only: conv_1d2nd

        character(*), parameter :: rout_name = 'reconstruct_PB3D_X_1'

        ! input / output
        type(modes_type), intent(in) :: mds                                  !< general modes variables
```

```
        type(grid_type), intent(in) :: grid_x                                   !< perturbation grid
        type(x_1_type), intent(inout) :: x                                      !< vectorial perturbation variables
        character(len=*), intent(in) :: data_name                               !< name to reconstruct
        integer, intent(in), optional :: rich_lvl                               !< Richardson level to reconstruct
        logical, intent(in), optional :: tot_rich                               !< whether to combine with previous Richardson levels
        integer, intent(in), optional :: lim_sec_x(2)                           !< limits of m_X (pol. flux) or n_X (tor. flux)
        integer, intent(in), optional :: lim_pos(3,2)                           !< position limits of subset of data

        ! local variables
        type(var_1d_type) :: var_1d                                             ! 1D variable
        integer :: lim_sec_x_loc(2)                                             ! local version of lim_sec_X
        integer :: lim_mem(4,2)                                                 ! memory limits for variables
        integer :: id                                                           ! counter
        integer :: rich_lvl_loc                                                 ! local rich_lvl
        integer :: par_id(3)                                                    ! parallel indices (start, end, stride)
        integer :: par_id_mem(2)                                                ! parallel indices (start, end) in memory (stride 1)
        integer :: par_lim(2)                                                   ! parallel limits
        integer :: rich_id(2)                                                   ! richardson level indices (start, end)

        ! initialize ierr
        ierr = 0

        ! set up local rich_lvl
        rich_lvl_loc = 0
        if (present(rich_lvl)) rich_lvl_loc = rich_lvl

        ! setup rich_id
        rich_id = setup_rich_id(rich_lvl_loc,tot_rich)

        ! set up local lim_sec_X
        lim_sec_x_loc = [1,n_mod_x]
        if (present(lim_sec_x)) lim_sec_x_loc = lim_sec_x

        ! set up parallel limits
        par_lim = [1,grid_x%n(1)]
        if (present(lim_pos)) then
            where (lim_pos(1,:).ge.0) par_lim = lim_pos(1,:)
        end if

        ! create X
        call x%init(mds,grid_x,lim_sec_x)

        ! restore looping over richardson levels
        do id = rich_id(2),rich_id(1),-1
            ! setup par_id
            par_id = setup_par_id(grid_x,rich_lvl_loc,id,tot_rich=tot_rich,&
                &par_lim=par_lim,par_id_mem=par_id_mem)

            ! set up local limits for HDF5 reconstruction
            lim_mem(1,:) = par_id_mem
            lim_mem(2,:) = [1,grid_x%n(2)]
            lim_mem(3,:) = [grid_x%i_min,grid_x%i_max]
            lim_mem(4,:) = lim_sec_x_loc
            if (present(lim_pos)) then
                lim_mem(2,:) = lim_pos(2,:)
                lim_mem(3,:) = lim_mem(3,:) + lim_pos(3,1) - 1                  ! take into account the grid limits (relative to position subset)
            end if

            ! RE_U_0
            ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                &'RE_U_0',rich_lvl=id,lim_loc=lim_mem)
            chckerr('')
            call conv_1d2nd(var_1d,dum_4d)
            x%U_0(par_id(1):par_id(2):par_id(3),:,:,:) = dum_4d
            call dealloc_var_1d(var_1d)

            ! IM_U_0
            ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                &'IM_U_0',rich_lvl=id,lim_loc=lim_mem)
            chckerr('')
            call conv_1d2nd(var_1d,dum_4d)
            x%U_0(par_id(1):par_id(2):par_id(3),:,:,:) = &
                x%U_0(par_id(1):par_id(2):par_id(3),:,:,:) + iu*dum_4d
            call dealloc_var_1d(var_1d)

            ! RE_U_1
            ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                &'RE_U_1',rich_lvl=id,lim_loc=lim_mem)
            chckerr('')
            call conv_1d2nd(var_1d,dum_4d)
            x%U_1(par_id(1):par_id(2):par_id(3),:,:,:) = dum_4d
            call dealloc_var_1d(var_1d)

```

```
            ! IM_U_1
            ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                &'IM_U_1',rich_lvl=id,lim_loc=lim_mem)
            chckerr('')
            call conv_1d2nd(var_1d,dum_4d)
            x%U_1(par_id(1):par_id(2):par_id(3),:,:,:) = &
                x%U_1(par_id(1):par_id(2):par_id(3),:,:,:) + iu*dum_4d
            call dealloc_var_1d(var_1d)

            ! RE_DU_0
            ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                &'RE_DU_0',rich_lvl=id,lim_loc=lim_mem)
            chckerr('')
            call conv_1d2nd(var_1d,dum_4d)
            x%DU_0(par_id(1):par_id(2):par_id(3),:,:,:) = dum_4d
            call dealloc_var_1d(var_1d)

            ! IM_DU_0
            ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                &'IM_DU_0',rich_lvl=id,lim_loc=lim_mem)
            chckerr('')
            call conv_1d2nd(var_1d,dum_4d)
            x%DU_0(par_id(1):par_id(2):par_id(3),:,:,:) = &
                x%DU_0(par_id(1):par_id(2):par_id(3),:,:,:) + iu*dum_4d
            call dealloc_var_1d(var_1d)

            ! RE_DU_1
            ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                &'RE_DU_1',rich_lvl=id,lim_loc=lim_mem)
            chckerr('')
            call conv_1d2nd(var_1d,dum_4d)
            x%DU_1(par_id(1):par_id(2):par_id(3),:,:,:) = dum_4d
            call dealloc_var_1d(var_1d)

            ! IM_DU_1
            ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                &'IM_DU_1',rich_lvl=id,lim_loc=lim_mem)
            chckerr('')
            call conv_1d2nd(var_1d,dum_4d)
            x%DU_1(par_id(1):par_id(2):par_id(3),:,:,:) = &
                x%DU_1(par_id(1):par_id(2):par_id(3),:,:,:) + iu*dum_4d
            call dealloc_var_1d(var_1d)
        end do
    end function reconstruct_pb3d_x_1

    !> Reconstructs the tensorial perturbation variables from PB3D HDF5 output.
    !!
    !! Also, if \c  rich_lvl is provided, <tt>_R_[rich_lvl]</tt>  is appended to
    !! the data name if it is > 0.
    !!
    !! With \c tot_rich  the information from previous Richardson  levels can be
    !! combined.
    !!
    !! Furthermore, using  \c lim_pos, you  can obtain a  subset of the  data by
    !! directly passing its  limits to the underlying HDF5  routine. This refers
    !! to the  position dimensions  only. If  provided, the  normal limits  of a
    !! divided grid refer to the subset, as in copy_grid().
    !!
    !! \c lim_sec_X, on the other hand, selects a range of mode numbers.
    !!
    !! \note The tensorial perturbation type can also be used for field- aligned
    !! variables, in which  case the first index is assumed  to have dimension 1
    !! only. This can be triggered using \c is_field_averaged.
    !!
    !! \return ierr
    integer function reconstruct_pb3d_x_2(mds,grid_X,X,data_name,rich_lvl,&
        &tot_rich,lim_sec_X,lim_pos,is_field_averaged) result(ierr)
        use num_vars, only: pb3d_name
        use hdf5_ops, only: read_hdf5_arr
        use pb3d_utilities, only: conv_1d2nd
        use x_utilities, only: get_sec_x_range

        character(*), parameter :: rout_name = 'reconstruct_PB3D_X_2'

        ! input / output
        type(modes_type), intent(in) :: mds                                     !< general modes variables
        type(grid_type), intent(in) :: grid_x                                   !< perturbation grid
        type(x_2_type), intent(inout) :: x                                      !< tensorial perturbation vars
        character(len=*), intent(in) :: data_name                               !< name to reconstruct
        integer, intent(in), optional :: rich_lvl                               !< Richardson level to reconstruct
        logical, intent(in), optional :: tot_rich                               !< whether to combine with previous
    Richardson levels
        integer, intent(in), optional :: lim_sec_x(2,2)                         !< limits of m_X (pol flux) or n_X (tor
    flux) for both dimensions
        integer, intent(in), optional :: lim_pos(3,2)                           !< position limits of subset of data
        logical, intent(in), optional :: is_field_averaged                      !< if field-averaged, only one dimension for
    first index
```

```fortran

        ! local variables
        type(var_1d_type) :: var_1d                                     ! 1D var
        logical :: is_field_averaged_loc                                ! local is_field_averaged
        integer :: id                                                   ! counter
        integer :: m, k                                                 ! counters
        integer :: sxr_loc(2,2)                                         ! local secondary X limits for symmetric and asymmetric vars
        integer :: sxr_tot(2,2)                                         ! total secondary X limits for symmetric and asymmetric vars
        integer :: nn_mod_loc(2)                                        ! local nr. of modes for symmetric and asymmetric vars
        integer :: lim_mem(4,2,2)                                       ! memory limits for variables for symmetric and asymmetric vars
        integer :: rich_lvl_loc                                         ! local rich_lvl
        integer :: par_id(3)                                            ! parallel indices (start, end, stride)
        integer :: par_id_mem(2)                                        ! parallel indices (start, end) in memory (stride 1)
        integer :: par_lim(2)                                           ! parallel limits
        integer :: rich_id(2)                                           ! richardson level indices (start, end)
        logical :: read_this(2)                                         ! whether symmetric and asymmetric variables need to be read

        ! initialize ierr
        ierr = 0

        ! set up local rich_lvl
        rich_lvl_loc = 0
        if (present(rich_lvl)) rich_lvl_loc = rich_lvl

        ! set up local is_field_averaged
        is_field_averaged_loc = .false.
        if (present(is_field_averaged)) is_field_averaged_loc = &
            &is_field_averaged

        ! setup rich_id
        rich_id = setup_rich_id(rich_lvl_loc,tot_rich)

        ! set up parallel limits
        par_lim = [1,grid_x%n(1)]
        if (present(lim_pos)) then
            where (lim_pos(1,:).ge.0) par_lim = lim_pos(1,:)
        end if

        ! create X
        call x%init(mds,grid_x,lim_sec_x,is_field_averaged)

        ! restore looping over richardson levels
        do id = rich_id(2),rich_id(1),-1
            ! setup par_id
            if (is_field_averaged_loc) then
                par_id = [1,1,1]                                        ! only first element
                par_id_mem = [1,1]
            else
                par_id = setup_par_id(grid_x,rich_lvl_loc,id,tot_rich=tot_rich,&
                    &par_lim=par_lim,par_id_mem=par_id_mem)
            end if

            ! loop over second dimension (horizontal)
            do m = 1,x%n_mod(2)
                ! get contiguous range of modes of this m
                call get_sec_x_range(sxr_loc(:,1),sxr_tot(:,1),m,.true.,&
                    &lim_sec_x)
                call get_sec_x_range(sxr_loc(:,2),sxr_tot(:,2),m,.false.,&
                    &lim_sec_x)
                nn_mod_loc = sxr_loc(2,:)-sxr_loc(1,:)+1
                read_this = .false.
                do k = 1,2
                    if (sxr_loc(1,k).le.sxr_loc(2,k)) read_this(k) = .true.     ! a bound is found
                end do

                ! set up local limits for HDF5 reconstruction of this m
                ! Note:  It  are  the  indices  in  total  matrix  sXr_tot  that
                ! correspond to the local limits.  "tot" just refers the to fact
                ! that they are valid for a  submatrix of the total matrix; They
                ! have been set up using the local grid_X limits as well.
                lim_mem(1,:,1) = par_id_mem
                lim_mem(2,:,1) = [1,grid_x%n(2)]
                lim_mem(3,:,1) = [grid_x%i_min,grid_x%i_max]
                lim_mem(4,:,1) = [sxr_tot(1,1),sxr_tot(2,1)]
                lim_mem(1,:,2) = par_id_mem
                lim_mem(2,:,2) = [1,grid_x%n(2)]
                lim_mem(3,:,2) = [grid_x%i_min,grid_x%i_max]
                lim_mem(4,:,2) = [sxr_tot(1,2),sxr_tot(2,2)]
                if (present(lim_pos)) then
                    lim_mem(2,:,1) = lim_pos(2,:)
                    lim_mem(3,:,1) = lim_mem(3,:,1) + lim_pos(3,1) - 1          ! take into account the grid limits
```

```
(relative to position subset)
                    lim_mem(2,:,2) = lim_pos(2,:)
                    lim_mem(3,:,2) = lim_mem(3,:,2) + lim_pos(3,1) - 1          ! take into account the grid limits
(relative to position subset)
                end if

                if (read_this(1)) then
                    ! RE_PV_0
                    ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                        &'RE_PV_0',rich_lvl=id,lim_loc=lim_mem(:,:,1))
                    chckerr('')
                    call conv_1d2nd(var_1d,dum_4d)
                    x%PV_0(par_id(1):par_id(2):par_id(3),:,:,&
                        &sxr_loc(1,1):sxr_loc(2,1)) = dum_4d
                    call dealloc_var_1d(var_1d)

                    ! IM_PV_0
                    ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                        &'IM_PV_0',rich_lvl=id,lim_loc=lim_mem(:,:,1))
                    chckerr('')
                    call conv_1d2nd(var_1d,dum_4d)
                    x%PV_0(par_id(1):par_id(2):par_id(3),:,:,&
                        &sxr_loc(1,1):sxr_loc(2,1)) = &
                        &x%PV_0(par_id(1):par_id(2):par_id(3),:,:,&
                        &sxr_loc(1,1):sxr_loc(2,1)) + iu*dum_4d
                    call dealloc_var_1d(var_1d)

                    ! RE_PV_2
                    ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                        &'RE_PV_2',rich_lvl=id,lim_loc=lim_mem(:,:,1))
                    chckerr('')
                    call conv_1d2nd(var_1d,dum_4d)
                    x%PV_2(par_id(1):par_id(2):par_id(3),:,:,&
                        &sxr_loc(1,1):sxr_loc(2,1)) = dum_4d
                    call dealloc_var_1d(var_1d)

                    ! IM_PV_2
                    ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                        &'IM_PV_2',rich_lvl=id,lim_loc=lim_mem(:,:,1))
                    chckerr('')
                    call conv_1d2nd(var_1d,dum_4d)
                    x%PV_2(par_id(1):par_id(2):par_id(3),:,:,&
                        &sxr_loc(1,1):sxr_loc(2,1)) = &
                        &x%PV_2(par_id(1):par_id(2):par_id(3),:,:,&
                        &sxr_loc(1,1):sxr_loc(2,1)) + iu*dum_4d
                    call dealloc_var_1d(var_1d)

                    ! RE_KV_0
                    ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                        &'RE_KV_0',rich_lvl=id,lim_loc=lim_mem(:,:,1))
                    chckerr('')
                    call conv_1d2nd(var_1d,dum_4d)
                    x%KV_0(par_id(1):par_id(2):par_id(3),:,:,&
                        &sxr_loc(1,1):sxr_loc(2,1)) = dum_4d
                    call dealloc_var_1d(var_1d)

                    ! IM_KV_0
                    ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                        &'IM_KV_0',rich_lvl=id,lim_loc=lim_mem(:,:,1))
                    chckerr('')
                    call conv_1d2nd(var_1d,dum_4d)
                    x%KV_0(par_id(1):par_id(2):par_id(3),:,:,&
                        &sxr_loc(1,1):sxr_loc(2,1)) = &
                        &x%KV_0(par_id(1):par_id(2):par_id(3),:,:,&
                        &sxr_loc(1,1):sxr_loc(2,1)) + iu*dum_4d
                    call dealloc_var_1d(var_1d)

                    ! RE_KV_2
                    ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                        &'RE_KV_2',rich_lvl=id,lim_loc=lim_mem(:,:,1))
                    chckerr('')
                    call conv_1d2nd(var_1d,dum_4d)
                    x%KV_2(par_id(1):par_id(2):par_id(3),:,:,&
                        &sxr_loc(1,1):sxr_loc(2,1)) = dum_4d
                    call dealloc_var_1d(var_1d)

                    ! IM_KV_2
                    ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                        &'IM_KV_2',rich_lvl=id,lim_loc=lim_mem(:,:,1))
                    chckerr('')
                    call conv_1d2nd(var_1d,dum_4d)
                    x%KV_2(par_id(1):par_id(2):par_id(3),:,:,&
                        &sxr_loc(1,1):sxr_loc(2,1)) = &
                        &x%KV_2(par_id(1):par_id(2):par_id(3),:,:,&
                        &sxr_loc(1,1):sxr_loc(2,1)) + iu*dum_4d
                    call dealloc_var_1d(var_1d)
                end if
```

```
                if (read_this(2)) then
                    ! RE_PV_1
                    ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                        &'RE_PV_1',rich_lvl=id,lim_loc=lim_mem(:,:,2))
                    chckerr('')
                    call conv_1d2nd(var_1d,dum_4d)
                    x%PV_1(par_id(1):par_id(2):par_id(3),:,:,&
                        &sxr_loc(1,2):sxr_loc(2,2)) = dum_4d
                    call dealloc_var_1d(var_1d)

                    ! IM_PV_1
                    ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                        &'IM_PV_1',rich_lvl=id,lim_loc=lim_mem(:,:,2))
                    chckerr('')
                    call conv_1d2nd(var_1d,dum_4d)
                    x%PV_1(par_id(1):par_id(2):par_id(3),:,:,&
                        &sxr_loc(1,2):sxr_loc(2,2)) = &
                        &x%PV_1(par_id(1):par_id(2):par_id(3),:,:,&
                        &sxr_loc(1,2):sxr_loc(2,2)) + iu*dum_4d
                    call dealloc_var_1d(var_1d)

                    ! RE_KV_1
                    ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                        &'RE_KV_1',rich_lvl=id,lim_loc=lim_mem(:,:,2))
                    chckerr('')
                    call conv_1d2nd(var_1d,dum_4d)
                    x%KV_1(par_id(1):par_id(2):par_id(3),:,:,&
                        &sxr_loc(1,2):sxr_loc(2,2)) = dum_4d
                    call dealloc_var_1d(var_1d)

                    ! IM_KV_1
                    ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                        &'IM_KV_1',rich_lvl=id,lim_loc=lim_mem(:,:,2))
                    chckerr('')
                    call conv_1d2nd(var_1d,dum_4d)
                    x%KV_1(par_id(1):par_id(2):par_id(3),:,:,&
                        &sxr_loc(1,2):sxr_loc(2,2)) = &
                        &x%KV_1(par_id(1):par_id(2):par_id(3),:,:,&
                        &sxr_loc(1,2):sxr_loc(2,2)) + iu*dum_4d
                    call dealloc_var_1d(var_1d)
                end if
            end do
        end do
    end function reconstruct_pb3d_x_2

    !> Reconstructs the vacuum variables from PB3D HDF5 output.
    !!
    !! Also, if \c  rich_lvl is provided, <tt>_R_[rich_lvl]</tt>  is appended to
    !! the data name if it is > 0.
    !!
    !! \return ierr
    integer function reconstruct_pb3d_vac(vac,data_name,rich_lvl) result(ierr)
        use num_vars, only: pb3d_name, rank, n_procs
        use hdf5_ops, only: read_hdf5_arr
        use pb3d_utilities, only: conv_1d2nd
        use x_utilities, only: get_sec_x_range

        character(*), parameter :: rout_name = 'reconstruct_PB3D_vac'

        ! input / output
        type(vac_type), intent(inout) :: vac                                    !< vacuum variables
        character(len=*), intent(in) :: data_name                               !< name to reconstruct
        integer, intent(in), optional :: rich_lvl                               !< Richardson level to reconstruct

        ! local variables
        type(var_1d_type) :: var_1d                                             ! 1D var
        integer :: rich_lvl_loc                                                 ! local rich_lvl
        integer :: style                                                        ! style of vacuum
        integer :: n_bnd                                                        ! number of terms in boundary
        integer :: prim_x                                                       ! primary mode number
        integer :: n_ang(2)                                                     ! number of angles (1) and number of field lines (2)
        real(dp) :: jq                                                          ! iota or q

        ! initialize ierr
        ierr = 0

        ! set up local rich_lvl
        rich_lvl_loc = 0
        if (present(rich_lvl)) rich_lvl_loc = rich_lvl

        ! misc_vac
        ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'misc_vac',&
            &rich_lvl=rich_lvl_loc)
        chckerr('')
        call conv_1d2nd(var_1d,dum_1d)
```

```
        style = nint(dum_1d(1))
        n_bnd = nint(dum_1d(2))
        prim_x = nint(dum_1d(3))
        n_ang = nint(dum_1d(4:5))
        jq = dum_1d(6)
        call dealloc_var_1d(var_1d)

        ! create vac
        ierr = vac%init(style,n_bnd,prim_x,n_ang,jq)
        chckerr('')

        ! secondary mode numbers
        ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
            &'sec_X',rich_lvl=rich_lvl_loc)
        chckerr('')
        call conv_1d2nd(var_1d,dum_1d)
        vac%sec_X = nint(dum_1d)
        deallocate(dum_1d)
        call dealloc_var_1d(var_1d)

        ! norm
        ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
            &'norm',rich_lvl=rich_lvl_loc)
        chckerr('')
        call conv_1d2nd(var_1d,dum_2d)
        vac%norm = dum_2d
        call dealloc_var_1d(var_1d)

        ! x_vec
        ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
            &'x_vec',rich_lvl=rich_lvl_loc)
        chckerr('')
        call conv_1d2nd(var_1d,dum_2d)
        vac%x_vec = dum_2d
        call dealloc_var_1d(var_1d)

        ! copy variables specific to style
        select case (vac%style)
            case (1)                                                    ! field-line 3-D
                ! h_fac
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                    &'h_fac',rich_lvl=rich_lvl_loc)
                chckerr('')
                call conv_1d2nd(var_1d,dum_2d)
                vac%h_fac = dum_2d
                call dealloc_var_1d(var_1d)
            case (2)                                                    ! axisymmetric
                ! ang
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                    &'ang',rich_lvl=rich_lvl_loc)
                chckerr('')
                call conv_1d2nd(var_1d,dum_2d)
                vac%ang = dum_2d
                call dealloc_var_1d(var_1d)

                ! dnorm
                ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                    &'dnorm',rich_lvl=rich_lvl_loc)
                chckerr('')
                call conv_1d2nd(var_1d,dum_2d)
                vac%dnorm = dum_2d
                call dealloc_var_1d(var_1d)
        end select

        if (rank.eq.n_procs-1) then
            ! RE_res
            ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                &'RE_res',rich_lvl=rich_lvl_loc)
            chckerr('')
            call conv_1d2nd(var_1d,dum_2d)
            vac%res = dum_2d
            call dealloc_var_1d(var_1d)

            ! IM_res
            ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),&
                &'IM_res',rich_lvl=rich_lvl_loc)
            chckerr('')
            call conv_1d2nd(var_1d,dum_2d)
            vac%res = vac%res + iu*dum_2d
            call dealloc_var_1d(var_1d)
        end if
    end function reconstruct_pb3d_vac

    !> Reconstructs the solution variables from PB3D HDF5 output.
    !!
    !! Also, if \c  rich_lvl is provided, <tt>_R_[rich_lvl]</tt>  is appended to
    !! the data name if it is > 0.
```

```
    !!
    !! Furthermore, using  \c lim_pos, you  can obtain a  subset of the  data by
    !! directly passing its  limits to the underlying HDF5  routine. This refers
    !! to the  position dimensions  only. If  provided, the  normal limits  of a
    !! divided grid refer to the subset, as in copy_grid().
    !!
    !! \c lim_sec_sol, on the other hand, selects a range of mode numbers.
    !!
    !! \return ierr
    integer function reconstruct_pb3d_sol(mds,grid_sol,sol,data_name,rich_lvl,&
        &lim_sec_sol,lim_pos) result(ierr)
        use num_vars, only: pb3d_name
        use hdf5_ops, only: read_hdf5_arr
        use x_vars, only: n_mod_x
        use pb3d_utilities, only: conv_1d2nd

        character(*), parameter :: rout_name = 'reconstruct_PB3D_sol'

        ! input / output
        type(modes_type), intent(in) :: mds                                     !< general modes variables
        type(grid_type), intent(in) :: grid_sol                                 !< solution grid
        type(sol_type), intent(inout) :: sol                                    !< solution variables
        character(len=*), intent(in) :: data_name                               !< name to reconstruct
        integer, intent(in), optional :: rich_lvl                               !< Richardson level to reconstruct
        integer, intent(in), optional :: lim_sec_sol(2)                         !< limits of m_X (pol. flux) or n_X (tor. flux)
        integer, intent(in), optional :: lim_pos(1,2)                           !< position limits of subset of data

        ! local variables
        type(var_1d_type) :: var_1d                                             ! 1D variable
        integer :: lim_sec_sol_loc(2)                                           ! local version of lim_sec_sol
        integer :: lim_mem(3,2)                                                 ! memory limits for variables
        integer :: n_ev                                                         ! nr. of Eigenvalues

        ! initialize ierr
        ierr = 0

        ! prepare

        ! set n_EV
        ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'RE_sol_vec',&
            &rich_lvl=rich_lvl)
        chckerr('')
        n_ev = var_1d%tot_i_max(3)-var_1d%tot_i_min(3)+1
        call dealloc_var_1d(var_1d)

        ! set up local lim_sec_sol
        lim_sec_sol_loc = [1,n_mod_x]
        if (present(lim_sec_sol)) lim_sec_sol_loc = lim_sec_sol

        ! create solution
        call sol%init(mds,grid_sol,n_ev,lim_sec_sol)

        ! set up local limits for HDF5 reconstruction
        lim_mem(1,:) = lim_sec_sol_loc
        lim_mem(2,:) = [grid_sol%i_min,grid_sol%i_max]
        lim_mem(3,:) = [-1,-1]
        if (present(lim_pos)) then
            lim_mem(2,:) = lim_mem(2,:) + lim_pos(1,1) - 1                      ! take into account the grid limits (relative to position subset)
        end if

        ! restore variables

        ! RE_sol_val
        ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'RE_sol_val',&
            &rich_lvl=rich_lvl)
        chckerr('')
        call conv_1d2nd(var_1d,dum_1d)
        sol%val = dum_1d
        deallocate(dum_1d)
        call dealloc_var_1d(var_1d)

        ! IM_sol_val
        ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'IM_sol_val',&
            &rich_lvl=rich_lvl)
        chckerr('')
        call conv_1d2nd(var_1d,dum_1d)
        sol%val = sol%val + iu*dum_1d
        deallocate(dum_1d)
        call dealloc_var_1d(var_1d)

        ! RE_sol_vec
        ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'RE_sol_vec',&
            &rich_lvl=rich_lvl,lim_loc=lim_mem)
        chckerr('')
        call conv_1d2nd(var_1d,dum_3d)
```

```
        sol%vec = dum_3d
        deallocate(dum_3d)
        call dealloc_var_1d(var_1d)

        ! IM_sol_vec
        ierr = read_hdf5_arr(var_1d,pb3d_name,trim(data_name),'IM_sol_vec',&
            &rich_lvl=rich_lvl,lim_loc=lim_mem)
        chckerr('')
        call conv_1d2nd(var_1d,dum_3d)
        sol%vec = sol%vec + iu*dum_3d
        deallocate(dum_3d)
        call dealloc_var_1d(var_1d)
    end function reconstruct_pb3d_sol

    !> get grid size
    !!
    !! \note <tt>grid_</tt> is added in front the grid_name.
    !!
    !! \return ierr
    integer function get_pb3d_grid_size(n,grid_name,rich_lvl,tot_rich) &
        &result(ierr)
        use num_vars, only: pb3d_name
        use pb3d_utilities, only: conv_1d2nd
        use hdf5_ops, only: read_hdf5_arr

        character(*), parameter :: rout_name = 'get_PB3D_grid_size'

        ! input / output
        integer, intent(inout) :: n(3)                                  !< n of grid
        character(len=*), intent(in) :: grid_name                       !< name of grid
        integer, intent(in), optional :: rich_lvl                       !< Richardson level to reconstruct
        logical, intent(in), optional :: tot_rich                       !< whether to combine with previous Richardson levels

        ! local variables
        type(var_1d_type) :: var_1d                                     ! 1D variable
        real(dp), allocatable :: dum_1d(:)                              ! dummy variables

        ! initialize ierr
        ierr = 0

        ! set n from HDF5 for local Richardson level
        n = 0
        ierr = read_hdf5_arr(var_1d,pb3d_name,'grid_'//trim(grid_name),&
            &'n',rich_lvl=rich_lvl)
        chckerr('')
        ! loop over all equilibrium jobs
        call conv_1d2nd(var_1d,dum_1d)
        n = nint(dum_1d)
        call dealloc_var_1d(var_1d)

        ! possibly only half of the points were saved in local Richardson level
        if (present(tot_rich) .and. present(rich_lvl)) then
            if (tot_rich .and. rich_lvl.gt.1) n(1) = n(1)*2+1
        end if
    end function get_pb3d_grid_size
end module pb3d_ops
```

D.66 /Users/toonweyens/Code/PB3D/Modules/PB3D_utilities.f90 File Reference

Interfaces and Types

- interface pb3d_utilities::conv_1d2nd

  Converts 1-D to n-D variables.

Modules

- module pb3d_utilities

  Numerical utilities related to PB3D operations.

Functions/Subroutines

- integer function, dimension(3), public pb3d_utilities::setup_par_id (grid, rich_lvl_max, rich_lvl_loc, tot_rich, par_lim, par_id_mem)

  Setup parallel id.
- integer function, dimension(2), public pb3d_utilities::setup_rich_id (rich_lvl_max, tot_rich)

  Returns richardson id.

D.67 PB3D_utilities.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Numerical utilities related to PB3D operations.
!-----------------------------------------------------------------------------!
module pb3d_utilities
#include <PB3D_macros.h>
    use str_utilities
    use messages
    use num_vars, only: dp, pi, max_str_ln, max_name_ln
    use hdf5_vars, only: var_1d_type

    implicit none
    private
    public conv_1d2nd, setup_rich_id, setup_par_id

    ! interfaces

    !> \public Converts 1-D to n-D variables.
    !!
    !! The 1-D variables are used for  internal storage in HDF5, whereas the n_D
    !! variables correspond to the ones in PB3D.
    interface conv_1d2nd
        !> \public
        module procedure conv_1d2nd_1d
        !> \public
        module procedure conv_1d2nd_2d
        !> \public
        module procedure conv_1d2nd_3d
        !> \public
        module procedure conv_1d2nd_4d
        !> \public
        module procedure conv_1d2nd_6d
        !> \public
        module procedure conv_1d2nd_7d
    end interface

contains
    !> \private 1-D version
    subroutine conv_1d2nd_1d(var_in,var_out)
        ! input / output
        type(var_1d_type), intent(in) :: var_in                                 !< 1D variable
        real(dp), intent(inout), allocatable :: var_out(:)                      !< output variable

        ! allocate and copy variable
        if (allocated(var_out)) deallocate(var_out)
        allocate(var_out(var_in%tot_i_min(1):var_in%tot_i_max(1)))
        var_out = reshape(var_in%p,shape(var_out))
    end subroutine conv_1d2nd_1d
    !> \private 2-D version
    subroutine conv_1d2nd_2d(var_in,var_out)
        ! input / output
        type(var_1d_type), intent(in) :: var_in                                 !< 1D variable
        real(dp), intent(inout), allocatable :: var_out(:,:)                    !< output variable

        ! allocate and copy variable
        if (allocated(var_out)) deallocate(var_out)
        allocate(var_out(&
            &var_in%tot_i_min(1):var_in%tot_i_max(1),&
            &var_in%tot_i_min(2):var_in%tot_i_max(2)))
        var_out = reshape(var_in%p,[&
            &var_in%tot_i_max(1)-var_in%tot_i_min(1)+1,&
            &var_in%tot_i_max(2)-var_in%tot_i_min(2)+1])
    end subroutine conv_1d2nd_2d
    !> \private 3-D version
    subroutine conv_1d2nd_3d(var_in,var_out)
        ! input / output
```

```
        type(var_1d_type), intent(in) :: var_in                                 !< 1D variable
        real(dp), intent(inout), allocatable :: var_out(:,:,:)                  !< output variable

        ! allocate and copy variable
        if (allocated(var_out)) deallocate(var_out)
        allocate(var_out(&
            &var_in%tot_i_min(1):var_in%tot_i_max(1),&
            &var_in%tot_i_min(2):var_in%tot_i_max(2),&
            &var_in%tot_i_min(3):var_in%tot_i_max(3)))
        var_out = reshape(var_in%p,shape(var_out))
    end subroutine conv_1d2nd_3d
    !> \private 4-D version
    subroutine conv_1d2nd_4d(var_in,var_out)
        ! input / output
        type(var_1d_type), intent(in) :: var_in                                 !< 1D variable
        real(dp), intent(inout), allocatable :: var_out(:,:,:,:)                !< output variable

        ! allocate and copy variable
        if (allocated(var_out)) deallocate(var_out)
        allocate(var_out(&
            &var_in%tot_i_min(1):var_in%tot_i_max(1),&
            &var_in%tot_i_min(2):var_in%tot_i_max(2),&
            &var_in%tot_i_min(3):var_in%tot_i_max(3),&
            &var_in%tot_i_min(4):var_in%tot_i_max(4)))
        var_out = reshape(var_in%p,shape(var_out))
    end subroutine conv_1d2nd_4d
    !> \private 6-D version
    subroutine conv_1d2nd_6d(var_in,var_out)
        ! input / output
        type(var_1d_type), intent(in) :: var_in                                 !< 1D variable
        real(dp), intent(inout), allocatable :: var_out(:,:,:,:,:,:)            !< output variable

        ! allocate and copy variable
        if (allocated(var_out)) deallocate(var_out)
        allocate(var_out(&
            &var_in%tot_i_min(1):var_in%tot_i_max(1),&
            &var_in%tot_i_min(2):var_in%tot_i_max(2),&
            &var_in%tot_i_min(3):var_in%tot_i_max(3),&
            &var_in%tot_i_min(4):var_in%tot_i_max(4),&
            &var_in%tot_i_min(5):var_in%tot_i_max(5),&
            &var_in%tot_i_min(6):var_in%tot_i_max(6)))
        var_out = reshape(var_in%p,shape(var_out))
    end subroutine conv_1d2nd_6d
    !> \private 7-D version
    subroutine conv_1d2nd_7d(var_in,var_out)
        ! input / output
        type(var_1d_type), intent(in) :: var_in                                 !< 1D variable
        real(dp), intent(inout), allocatable :: var_out(:,:,:,:,:,:,:)          !< output variable

        ! allocate and copy variable
        if (allocated(var_out)) deallocate(var_out)
        allocate(var_out(&
            &var_in%tot_i_min(1):var_in%tot_i_max(1),&
            &var_in%tot_i_min(2):var_in%tot_i_max(2),&
            &var_in%tot_i_min(3):var_in%tot_i_max(3),&
            &var_in%tot_i_min(4):var_in%tot_i_max(4),&
            &var_in%tot_i_min(5):var_in%tot_i_max(5),&
            &var_in%tot_i_min(6):var_in%tot_i_max(6),&
            &var_in%tot_i_min(7):var_in%tot_i_max(7)))
        var_out = reshape(var_in%p,shape(var_out))
    end subroutine conv_1d2nd_7d

    !> Setup parallel id.
    !!
    !! The parallel id consists of:
    !!  - \c par_id(1): start index
    !!  - \c par_id(2): end index
    !!  - \c par_id(3): stride
    !!
    !! These are set  up by considering the transformation between  a point \f$r
    !! \in  (1\ldots n)\f$  in the  local variable,  with corresponding  indices
    !! \f$(a\ldots b)\f$ indicated by \c par_lim in the HDF5 variable in memory,
    !! so that \f$n = b-a+1\f$:
    !!   \f[p + k s = a + r - 1,\f]
    !! where \f$k\f$ is  an integer, \f$s\f$ is the stride  for Richardson level
    !! \f$i\f$ (with max. level \f$I\f$):
    !!  \f[\begin{aligned}
    !!   s &= 2^{I-1}   \  \text{for} \ i = 1 \\
    !!     &= 2^{I-i+1} \  \text{for} \ i > 1
    !!  \end{aligned}\f]
    !! and where \f$p\f$ is the starting point in the HDF5 variables:
    !!  \f[\begin{aligned}
    !!   p &= 1         \  \text{for} \ i = 1 \\
    !!     &= 1 + s/2   \  \text{for} \ i > 1
    !!  \end{aligned}\f]
    !!
    !! Therefore, the minimum possible \f$r\f$ lies in \f$(0..s-1)\f$:
```

```
    !!  \f[\mod(r-1,s) = r_\text{min}-1, \f]
    !! which leads to
    !!  \f[r_\text{min} = 1 + \mod(p-a,s). \f]
    !!
    !! The maximum possible \f$r\f$, then, lies in \f$(n-s+1..n)\f$, which leads
    !! to:
    !!  \f[ r_\text{max} = n - s + 1 + \mod(p-b-1,s). \f]
    !!
    !! These limits  and the  strides are  saved in \c  par_id =  \f$\vec{r} =
    !! \begin{pmatrix}r_\text{min}\\ r_\text{max}\end{pmatrix}\f$.
    !!
    !! If  the optional  indices  \f$a\f$ and  \f$b\f$ are  not  given they  are
    !! assumed to  be 1 and \f$n\f$,  with \f$n = 1+ks\f$,  which simplifies the
    !! equations to:
    !!  \f[\begin{aligned}
    !!   r_\text{min} &= p \\
    !!   r_\text{max} &= n-p+1 .
    !!  \end{aligned}\f]
    !!
    !! Optionally,  the  indices  in  the  HDF5  arrays  can  also  be
    !! returned  in  \c  par_id_mem  =  \f$\begin{pmatrix}R_\text{min}\\
    !! R_\text{max}\end{pmatrix}\f$. These are equal to \f$k-1\f$, where \f$k\f$
    !! is the integer refered to above:
    !!  \f[\vec{R} = 1 + \frac{a-1-p+\vec{r}}{s}, \f]
    !! where the addition  between a vector and  a scalar should be  seen as the
    !! element-wise operation.
    function setup_par_id(grid,rich_lvl_max,rich_lvl_loc,tot_rich,par_lim,&
        &par_id_mem) result(par_id)
        use grid_vars, only: grid_type

        ! input / output
        type(grid_type), intent(in) :: grid                                 !< grid
        integer, intent(in) :: rich_lvl_max                                 !< maximum Richardson level
        integer, intent(in) :: rich_lvl_loc                                 !< local Richardson level
        logical, intent(in), optional :: tot_rich                           !< whether to combine with previous
    Richardson levels
        integer, intent(in), optional :: par_lim(2)                         !< limits (a..b) of variable requested
        integer, intent(inout), optional :: par_id_mem(2)                   !< parallel id in memory
        integer :: par_id(3)                                                !< parallel id

        ! local variables
        integer :: s                                                        ! stride
        integer :: p                                                        ! lower limit in memory
        integer :: n                                                        ! b-a+1
        logical :: tot_rich_loc                                             ! local tot_rich
        integer :: par_lim_loc(2)                                           ! local par_lim

        ! set up local tot_rich and par_lim
        tot_rich_loc = .false.
        if (present(tot_rich) .and. rich_lvl_max.gt.1) tot_rich_loc = tot_rich  ! only for higher Richardson levels
        par_lim_loc = [1,grid%n(1)]
        if (present(par_lim)) par_lim_loc = par_lim

        ! set up parallel indices
        n = par_lim_loc(2)-par_lim_loc(1)+1
        if (tot_rich_loc) then
            if (rich_lvl_loc.eq.1) then
                s = 2**(rich_lvl_max-1)
                p = 1
            else
                s = 2**(rich_lvl_max+1-rich_lvl_loc)
                p = 1 + s/2
            end if
        else
            s = 1
            p = 1
        end if
        par_id(1) = 1 + modulo(p-par_lim_loc(1),s)
        par_id(2) = n - s + 1 + modulo(p-par_lim_loc(2)-1,s)
        par_id(3) = s

        ! set par_id in memory if requested
        if (present(par_id_mem)) par_id_mem = &
            &1 + (par_lim_loc(1)-1-p+par_id(1:2))/s
    end function setup_par_id

    !> Returns richardson id.
    !!
    !! \c rich_id has the following structure:
    !!  - \c rich_id(1): start Richardson level
    !!  - \c rich_id(2): end Richardson level
    function setup_rich_id(rich_lvl_max,tot_rich) result(rich_id)
        ! input / output
        integer, intent(in) :: rich_lvl_max                                 !< maximum Richardson level
        logical, intent(in), optional :: tot_rich                           !< whether to combine with previous
    Richardson levels
        integer :: rich_id(2)                                               !< Richardson id
```

```

        ! set up rich_id
        rich_id = [rich_lvl_max,rich_lvl_max]
        if (present(tot_rich) .and. rich_lvl_max.gt.1) then                    ! only for higher Richardson levels
            if (tot_rich) rich_id = [1,rich_lvl_max]
        end if
    end function setup_rich_id
end module pb3d_utilities
```

D.68 /Users/toonweyens/Code/PB3D/Modules/rich_ops.f90 File Reference

Modules

- module rich_ops

  Operations concerning Richardson extrapolation.

Functions/Subroutines

- integer function, public rich_ops::init_rich ()

  Initialize Richardson extrapolation system.
- subroutine, public rich_ops::term_rich ()

  Terminate the Richardson extrapolation system.
- logical function, public rich_ops::do_rich ()

  Tests whether this Richardson level should be done.
- integer function, public rich_ops::start_rich_lvl ()

  Start a Richardson level.
- subroutine, public rich_ops::stop_rich_lvl ()

  Stop a Richardson level.
- subroutine, public rich_ops::calc_rich_ex (sol_val)

  Calculates the coefficients of the Eigenvalues in the Richardson extrapolation.
- integer function, public rich_ops::find_max_rich_lvl (lvl_rich_req, max_lvl_rich_file)

  Probe to find out which Richardson levels are available.

D.69 rich_ops.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Operations concerning Richardson extrapolation.
!-----------------------------------------------------------------------------!
module rich_ops
#include <PB3D_macros.h>
#include <wrappers.h>
    use str_utilities
    use output_ops
    use messages
    use num_vars, only: dp, pi, max_str_ln, max_it_rich, tol_rich
    use grid_vars, only: grid_type
    use eq_vars, only: eq_1_type
    use sol_vars, only: sol_type
    use rich_vars

    implicit none
    private
    public init_rich, term_rich, start_rich_lvl, stop_rich_lvl, do_rich, &
        &calc_rich_ex, find_max_rich_lvl
```

```fortran

contains
    !> Initialize Richardson extrapolation system.
    !!
    !! Needs to be called only once.
    !!
    !! \return ierr
    integer function init_rich() result(ierr)
        use num_vars, only: n_sol_requested, rich_restart_lvl, eq_style, rank
        use pb3d_ops, only: reconstruct_pb3d_in, reconstruct_pb3d_grid, &
            &reconstruct_pb3d_eq_1, reconstruct_pb3d_sol
        use x_ops, only: init_modes, setup_modes
        use x_vars, only: mds_sol
        use grid_vars, only: n_alpha, min_alpha, max_alpha, alpha
        use grid_utilities, only: calc_eqd_grid

        character(*), parameter :: rout_name = 'init_rich'

        ! local variables
        type(grid_type) :: grid_eq_b                                    !< equilibrium grid
        type(grid_type) :: grid_x_b                                     !< field-aligned perturbation grid
        type(grid_type) :: grid_sol                                     !< solution grid
        type(eq_1_type) :: eq                                           !< flux equilibrium variables
        type(sol_type) :: sol                                           !< solution variables
        character(len=4) :: grid_eq_name                                !< name of equilibrium grid
        character(len=4) :: grid_x_name                                 !< name of perturbation grid
        character(len=max_str_ln) :: err_msg                            !< error message

        ! set variables
        rich_lvl = 1
        rich_conv = .false.
        use_guess = .false.

        ! initialize ierr
        ierr = 0

        ! user output (only master has variables set up)
        if (rank.eq.0 .and. max_it_rich.gt.1) then                      ! only when more than one level
            if (rich_restart_lvl.eq.1) then
                ! user output
                call writo('Richardson extrapolation loop initialization')
            else
                ! user output
                call writo('Richardson extrapolation loop continuation at &
                    &level '//trim(i2str(rich_restart_lvl)))
            end if
        end if
        call lvl_ud(1)

        ! some preliminary things
        ierr = reconstruct_pb3d_in('in')                                ! reconstruct miscellaneous PB3D output variables
        chckerr('')

        ! allocate variables
        allocate(sol_val_rich(max_it_rich,max_it_rich,n_sol_requested))
        sol_val_rich = 0.0_dp
        allocate(x_axis_rich(max_it_rich,n_sol_requested))
        allocate(max_rel_err(max_it_rich-1))
        allocate(loc_max_rel_err(max_it_rich-1,1))

        ! set up alpha
        allocate(alpha(n_alpha))
        ierr = calc_eqd_grid(alpha,min_alpha*pi,max_alpha*pi,excl_last=.true.)
        chckerr('')

        ! Richardson restart
        if (rich_restart_lvl.gt.1) then
            ! user output
            call writo('reconstructing Richardson extrapolation variables')
            call lvl_ud(1)

            call writo('Prepare general variables')
            call lvl_ud(1)

            ! use guess
            use_guess = .true.

            ! set up grid names
            select case (eq_style)
                case (1)                                                ! VMEC
                    grid_x_name = 'X'                                   ! grids are already field-aligned
                    grid_eq_name = 'eq'                                 ! grids are already field-aligned
                case (2)                                                ! HELENA
                    grid_x_name = 'X_B'                                 ! field-aligned grid is separate
                    grid_eq_name = 'eq_B'                               ! field-aligned grid is separate
            end select
```

```fortran

            ! reconstruct field-aligned equilibrium grid (for level 1)
            ierr = reconstruct_pb3d_grid(grid_eq_b,trim(grid_eq_name),&
                &rich_lvl=1)
            chckerr('')

            ! reconstruct solution grid
            ierr = reconstruct_pb3d_grid(grid_sol,'sol')
            chckerr('')

            ! reconstruct flux equilibrium variables
            ierr = reconstruct_pb3d_eq_1(grid_eq_b,eq,'eq_1')
            chckerr('')

            call lvl_ud(-1)

            ! loop  over all  previous Richardson  levels using  rich_lvl as
            ! counter
            do rich_lvl = 1,rich_restart_lvl-1
                call writo('Prepare variables for Richardson level '//&
                    &trim(i2str(rich_lvl)))
                call lvl_ud(1)

                ! reconstruct field-aligned perturbation grid
                ierr = reconstruct_pb3d_grid(grid_x_b,trim(grid_x_name),&
                    &rich_lvl=rich_lvl)
                chckerr('')

                if (rich_lvl.eq.1) then
                    ! initialize global n and m variables
                    ierr = init_modes(grid_eq_b,eq)
                    chckerr('')

                    ! set up n and m variables for sol grid
                    ierr = setup_modes(mds_sol,grid_eq_b,grid_sol)
                    chckerr('')
                end if

                ! reconstruct solution
                ierr = reconstruct_pb3d_sol(mds_sol,grid_sol,sol,'sol',&
                    &rich_lvl=rich_lvl)
                chckerr('')

                call lvl_ud(-1)

                call writo('Recreate Richardson variables')
                call lvl_ud(1)

                ! set x_axis_rich
                if (rich_lvl.eq.1) then
                    ! set up n_par_X for first rich_lvl
                    n_par_x = grid_x_b%n(1)

                    ! check it
                    if (n_par_x.eq.min_n_par_x) then
                        ! set x_axis_rich
                        x_axis_rich(1,:) = min_n_par_x
                    else
                        ierr = 1
                        err_msg = 'Saved variables started with &
                            &min_n_par_X = '//trim(i2str(n_par_x))//&
                            &' whereas here '//trim(i2str(min_n_par_x))//&
                            &' is specified'
                        chckerr(err_msg)
                    end if
                else
                    ! update n_par_X
                    n_par_x = n_par_x*2-1

                    ! set x_axis_rich
                    x_axis_rich(rich_lvl,:) = x_axis_rich(rich_lvl-1,:)*2-1
                end if

                ! calculate Richardson extrapolation
                call calc_rich_ex(sol%val)

                ! update error variables
                call check_conv()

                ! clean up
                call grid_x_b%dealloc()
                call sol%dealloc()

                call lvl_ud(-1)
            end do

            ! clean up
```

```
            call mds_sol%dealloc()
            call grid_eq_b%dealloc()
            call grid_sol%dealloc()
            call eq%dealloc()

            ! set rich_lvl
            rich_lvl = rich_restart_lvl

            call lvl_ud(-1)
            call writo('Richardson extrapolation variables reconstructed')
        end if

        if (max_it_rich.gt.1) then                                          ! only when more than one level
            call writo('Maximum number of iterations: '//&
                &trim(i2str(max_it_rich)))
            call writo('Tolerance requested: '//trim(r2strt(tol_rich)))

            call writo('')
        end if

        call lvl_ud(-1)
    end function init_rich

    !> Terminate the Richardson extrapolation system.
    !!
    !! Needs to be called only once.
    subroutine term_rich()
        use num_vars, only: rank, norm_disc_prec_sol
        use input_utilities, only: dealloc_in

        ! local variables
        integer :: id                                                       ! counter

        ! user output
        if (max_it_rich.gt.1) then
            ! user output
            call writo('Finishing Richardson extrapolation loop')
            call lvl_ud(1)

            if (rich_conv) then
                call writo('Convergence reached in '//trim(i2str(rich_lvl-1))//&
                    &' steps:')
            else
                call writo('After '//trim(i2str(rich_lvl-1))//&
                    &' steps, no convergence reached:')
            end if

            call lvl_ud(1)

            call writo('Maximum relative error '//&
                &trim(r2str(max_rel_err(rich_lvl-2)))//&
                &' for Eigenvalue '//trim(i2str(loc_max_rel_err(rich_lvl-2,1))))
            call writo('Tolerance requested: '//trim(r2strt(tol_rich)))
            call writo('Resulting best guesses for Eigenvalues:')
            call lvl_ud(1)
            do id = 1,size(sol_val_rich,3)
                call writo('For Eigenvalue '//trim(i2str(id))//':')
                call lvl_ud(1)
                call writo(trim(c2str(sol_val_rich(rich_lvl-1,rich_lvl-1,id))),&
                    &alert=.true.)
                call lvl_ud(-1)
            end do
            call lvl_ud(-1)
            call writo('with, theoretically, an error of the order O(Δ^'//&
                &trim(i2str(2*norm_disc_prec_sol*rich_lvl))//'),')
            call writo('compared to O(Δ^'//trim(i2str(2*norm_disc_prec_sol))//&
                &') without Richardson extrapolation')
            call writo('Notice that this is ONLY valid if the problem in the &
                &different Richardson stages is similar:')
            call lvl_ud(1)
            call writo('- It has to be part of the point spectrum')
            call writo('- The Eigenvectors should look similar')
            call lvl_ud(-1)

            call lvl_ud(-1)

            ! draw X values
            if (rank.eq.0) call draw_sol_val_rich()

            call writo('')

            call lvl_ud(-1)
        end if

        ! clean up
        call dealloc_in()
    contains
```

```
        ! Draws the Eigenvalues for the different levels of Richardson
        ! extrapolation as a function of the number of parallel points.
        !> \private
        subroutine draw_sol_val_rich()
            use num_vars, only: n_sol_requested

            ! local variables
            character(len=max_str_ln) :: plot_title                          ! title for plots
            character(len=max_str_ln) :: plot_name                           ! name of plot

            ! user output
            call writo('Plotting Eigenvalues as function of nr. of parallel &
                &points')
            call lvl_ud(1)

            ! print, using rich_lvl-1 as it has been aumented
            plot_title = 'Eigenvalues as function of nr. of parallel points'
            plot_name = 'Eigenvalues_richardson'
            call print_ex_2d([plot_title],plot_name,&
                &rp(sol_val_rich(1:rich_lvl-1,1,:)),&
                &x=x_axis_rich(1:rich_lvl-1,:),draw=.false.)

            ! output in file
            call draw_ex([plot_title],plot_name,n_sol_requested,1,.false.)

            ! user output
            call lvl_ud(-1)
            call writo('Done plotting Eigenvalues')
        end subroutine draw_sol_val_rich
    end subroutine term_rich

    !> Tests whether this Richardson level should be done.
    logical function do_rich()
        if (rich_lvl.gt.max_it_rich .or. rich_conv) then
            do_rich = .false.
        else
            do_rich = .true.
        end if
    end function do_rich

    !> Start a Richardson level.
    !!
    !! Calculates \c n_par_X for this level.  Then uses this to divide the jobs.
    !! equilibrium Finally, the limits are set up for these jobs.
    !!
    !! \return ierr
    integer function start_rich_lvl() result(ierr)
        use grid_utilities, only: calc_n_par_x_rich
        use grid_vars, only: n_r_eq, n_r_sol, n_alpha
        use eq_ops, only: divide_eq_jobs, calc_eq_jobs_lims
        use helena_vars, only: nchi
        use num_vars, only: eq_style, jump_to_sol, rich_restart_lvl
        use mpi_utilities, only: get_ser_var

        character(*), parameter :: rout_name = 'start_rich_lvl'

        ! local variables
        integer :: n_div                                                     !< number of divisions in equilibrium jobs
        logical :: only_half_grid                                            !< calculate only half grid with even points
        integer :: n_par_x_loc                                               !< local n_par_X
        integer :: var_size_without_par(2)                                   !< size without parallel dimension for eq_2
and X_1 variables

        ! initialize ierr
        ierr = 0

        ! set use_guess to .false. if user sets no_guess
        if (no_guess) use_guess = .false.

        ! Calculate total number of parallel points for the solution in
        ! Richardson loops
        if (rich_lvl.eq.1) then
            n_par_x = min_n_par_x
            only_half_grid = .false.
        else
            n_par_x = 2 * n_par_x - 1
            only_half_grid = .true.
        end if

        ! use calc_n_par_X_rich, taking into account possible half grid
        ierr = calc_n_par_x_rich(n_par_x_loc,only_half_grid)
        chckerr('')

        if (rich_lvl.eq.rich_restart_lvl .and. jump_to_sol) then
            call writo('Jumping straight to solution, no need to divide into &
                &equilibrium jobs')

```

```
            return
        end if

        ! Get local n_par_X to divide the equilibrium jobs. Note that the
        ! average size of eq_2 variables for all processes together is n_r_eq,
        ! times the number of field lines, the size due to the dimensions
        ! corresponding to the derivatives, and the number of variables (see
        ! subroutine 'calc_memory_eq' and 'calc_memory_X'), while it is n_r_sol
        ! for the X_1 variables.
        var_size_without_par(1) = n_r_eq
        var_size_without_par(2) = n_r_sol
        var_size_without_par = var_size_without_par * n_alpha
        select case (eq_style)
            case (1)                                                    ! VMEC
                ! divide equilibrium jobs
                ierr = divide_eq_jobs(n_par_x_loc,var_size_without_par,&
                    &n_div)
                chckerr('')
            case (2)                                                    ! HELENA
                ! divide equilibrium jobs
                ! Note: calculations for first Richardson level have to be done
                ! with a single job, as this is the calculation where the
                ! variables are calculated that are interpolated in all levels.
                ierr = divide_eq_jobs(nchi,var_size_without_par,n_div,&
                    &n_div_max=1,range_name='poloidal points in the &
                    &axisymmetric HELENA cross-section')                ! everything is tabulated on nchi poloidal points
                chckerr('')
        end select

        ! calculate equilibrium job limits
        ierr = calc_eq_jobs_lims(n_par_x_loc,n_div)
        chckerr('')
    end function start_rich_lvl

    !> Stop a Richardson level.
    !!
    !! It decides whether convergence has been reached, and possibly sets a
    !! guess for the next level.
    subroutine stop_rich_lvl()
        ! update the x axis of the Eigenvalue plot
        x_axis_rich(rich_lvl,:) = 1._dp*n_par_x

        ! Richardson extrapolation
        if (max_it_rich.gt.1) then                                      ! only do this if more than 1 Richardson level
            ! user output
            if (rich_lvl.gt.1) call writo('Richardson level '//&
                &trim(i2str(rich_lvl))//' summary')
            call lvl_ud(1)

            ! decide whether converged or not
            call check_conv()

            ! setup possible guess for next Richardson level
            call set_guess()

            if (rich_lvl.gt.1) call writo('')
            call lvl_ud(-1)
        else                                                            ! if not, Richardson is done
            rich_conv = .true.
        end if

        ! increase level
        rich_lvl = rich_lvl + 1
    contains
        ! Decides whether a guess should be used in a possible next Richardson
        ! level.
        !> \private
        subroutine set_guess()
            ! local variables
            real(dp), parameter :: tol = 5._dp                          ! tolerance for increase of error

            ! decide on guess
            if (rich_lvl.ge.3) then                                     ! start from level 3
                use_guess = max_rel_err(rich_lvl-1).lt.&
                    &tol*max_rel_err(rich_lvl-2)                        ! set guess for next level to true if max_rel_err is decreasing
            else
                use_guess = .true.                                      ! for first Richardson level, set guess for next level to true
            end if
        end subroutine set_guess
    end subroutine stop_rich_lvl

    !> Calculates the coefficients of the Eigenvalues in the Richardson
    !! extrapolation.
```

```fortran
    !!
    !! This is done using the recursive formula:
    !!  \f[ s(l,i,:) = s(l,i-1,:) +  \frac{1}{2^{2 \ p \ i} - 1}
    !!       (s(l,i-1,:) - s(l-1,i-1,:)),
    !!  \f]
    !! where  \f$s\f$  =  \c  sol_val_rich,  \f$i\f$  =  \c  ir,  \f$l\f$  =  \c
    !! lvl  and \f$p\f$  = \c  norm_disc_prec_sol,  the order  of the  numerical
    !! discretization. as  described in \cite  dahlquist2003numerical, algorithm
    !! 3.2, p. 306.
    subroutine calc_rich_ex(sol_val)
        use num_vars, only: norm_disc_prec_sol

        ! input / output
        complex(dp), intent(in) :: sol_val(:)                                   !< EV for this Richardson level

        ! local variables
        integer :: ir                                                           ! counter

        ! do calculations if rich_lvl > 1
        sol_val_rich(rich_lvl,1,1:size(sol_val)) = sol_val                      ! size(sol_val) can be less than n_sol_requested
        do ir = 2,rich_lvl
            sol_val_rich(rich_lvl,ir,1:size(sol_val)) = &
                &sol_val_rich(rich_lvl,ir-1,1:size(sol_val)) + &
                &1._dp/(2**(2*(ir-1)*norm_disc_prec_sol)-1._dp) * &
                &(sol_val_rich(rich_lvl,ir-1,1:size(sol_val)) - &
                &sol_val_rich(rich_lvl-1,ir-1,1:size(sol_val)))
        end do
    end subroutine calc_rich_ex

    !> Decides  whether   convergence  has  been  reached   for  the  Richardson
    !! extrapolation.
    !!
    !! This is done by checking whether the difference between the approximation
    !! of the Eigenvalues  in this Richardson level and  the previous Richardson
    !! level, with the same order of the error, falls below a threshold.
    !!
    !! Also increases the next tol_SLEPC if it is too low.
    !!
    !! \see calc_rich_ex()
    subroutine check_conv()
        use num_vars, only: tol_slepc

        ! local variables
        real(dp), allocatable :: corr(:)                                        ! correction for order (rich_lvl-1) of error
        integer :: ir                                                           ! counter
        real(dp) :: tol_slepc_adapt_factor = 0.01_dp                            ! factor to relate tol_SLEPC to maximum relative error

        if (rich_lvl.gt.1) then                                                 ! only do this if in Richardson level higher than 1
            ! set relative correction
            allocate(corr(size(sol_val_rich,3)))
            corr = abs(2*(sol_val_rich(rich_lvl,rich_lvl-1,:)-&
                &sol_val_rich(rich_lvl-1,rich_lvl-1,:))/&
                &(sol_val_rich(rich_lvl,rich_lvl-1,:)+&
                &sol_val_rich(rich_lvl-1,rich_lvl-1,:)))

            ! get maximum and location of maximum for relative correction
            max_rel_err(rich_lvl-1) = maxval(corr)
            loc_max_rel_err(rich_lvl-1,:) = maxloc(corr)

            ! user output
            do ir = 1,rich_lvl-1
                call writo('Richardson level '//trim(i2str(ir))//' -> '//&
                    &trim(i2str(ir+1))//': maximum relative error '//&
                    &trim(r2strt(max_rel_err(ir)))//' for Eigenvalue '//&
                    &trim(i2str(loc_max_rel_err(ir,1))))
            end do

            ! check whether tolerance has been reached
            if (max_rel_err(rich_lvl-1).lt.tol_rich) then
                rich_conv = .true.
                call writo('tolerance '//trim(r2strt(tol_rich))//&
                    &' reached after '//trim(i2str(rich_lvl))//&
                    &' iterations')
            else
                call writo('tolerance '//trim(r2strt(tol_rich))//' not yet &
                    &reached')
            end if

            ! check whether to decrease next tol_SLEPC
            if (rich_lvl.lt.max_it_rich) tol_slepc(rich_lvl+1) = min(&
                &tol_slepc(rich_lvl+1),&
                &tol_slepc_adapt_factor*max_rel_err(rich_lvl-1))
        end if
    end subroutine check_conv
```

```
    !> Probe to find out which Richardson levels are available.
    !!
    !! \return ierr
    integer function find_max_rich_lvl(lvl_rich_req,max_lvl_rich_file) &
        &result(ierr)
        use num_vars, only: pb3d_name
        use hdf5_utilities, only: probe_hdf5_group

        character(*), parameter :: rout_name = 'find_max_rich_lvl'

        ! input / output
        integer, intent(inout) :: lvl_rich_req                                  !< requested Richardson level
        integer, intent(inout) :: max_lvl_rich_file                             !< max. Richardson level found in file

        ! local variables
        integer :: ir                                                           ! counter
        character(len=max_str_ln) :: group_name                                 ! name of group to probe for
        logical :: group_exists                                                 ! whether probed group exists

        ! initialize ierr
        ierr = 0

        ! try openining solution for different Richardson extrapolation levels
        group_exists = .true.                                                   ! group_exists becomes stopping criterion
        ir = 1                                                                  ! initialize counter
        if (lvl_rich_req.gt.0 .and. lvl_rich_req.lt.huge(1)) ir = lvl_rich_req  ! move to requested
        do while (group_exists)
            group_name = 'sol_R_'//trim(i2str(ir))
            ierr = probe_hdf5_group(pb3d_name,group_name,group_exists)
            chckerr('')
            ir = ir + 1                                                         ! increment counter
        end do
        max_lvl_rich_file = ir-2                                                ! -2 because there will be one additional iteration
    end function find_max_rich_lvl
end module rich_ops
```

D.70 /Users/toonweyens/Code/PB3D/Modules/rich_vars.f90 File Reference

Modules

- module rich_vars

  Variables concerning Richardson extrapolation.

Functions/Subroutines

- elemental character(len=max_str_ln) function, public rich_vars::rich_info ()

  Returns string with possible extension with Richardson level or nothing if only one level and one parallel job.

Variables

- integer, public rich_vars::rich_lvl

  current level of Richardson extrapolation
- integer, public rich_vars::n_par_x

  nr. of parallel points in field-aligned grid
- integer, parameter, public rich_vars::req_min_n_par_x = 20

  required minimum n_par_X
- integer, public rich_vars::min_n_par_x

  min. of n_par_X (e.g. first value in Richardson loop)
- logical, public rich_vars::use_guess

whether a guess is formed from previous level of Richardson

- logical, public rich_vars::no_guess

  disable guessing Eigenfunction from previous level of Richardson
- logical, public rich_vars::rich_conv

  if Richarson extrapolation has converged
- complex(dp), dimension(:,:,:), allocatable, public rich_vars::sol_val_rich

  Richardson array of eigenvalues.
- real(dp), dimension(:,:), allocatable, public rich_vars::x_axis_rich

  x axis for plot of Eigenvalues with Richardson level
- real(dp), dimension(:), allocatable, public rich_vars::max_rel_err

  maximum relative error for all Richardson levels
- integer, dimension(:,:), allocatable, public rich_vars::loc_max_rel_err

  location of maximum of relative error

D.71 rich_vars.f90

Go to the documentation of this file.

```
!-------------------------------------------------------------------------!
!> Variables concerning Richardson extrapolation.
!-------------------------------------------------------------------------!
module rich_vars
#include <PB3D_macros.h>
    use str_utilities
    use output_ops
    use messages
    use num_vars, only: dp, pi, max_str_ln, max_it_rich, tol_rich

    implicit none
    private
    public rich_info, &
        &rich_lvl, no_guess, use_guess, n_par_x, min_n_par_x, rich_conv, &
        &sol_val_rich, x_axis_rich, max_rel_err, loc_max_rel_err, &
        &req_min_n_par_x

    ! global variables
    integer :: rich_lvl                                                 !< current level of Richardson extrapolation
    integer :: n_par_x                                                  !< nr. of parallel points in field-aligned grid
    integer, parameter :: req_min_n_par_x = 20                          !< required minimum \c n_par_X
    integer :: min_n_par_x                                              !< min. of \c n_par_X (e.g. first value in Richardson loop)
    logical :: use_guess                                                !< whether a guess is formed from previous level of Richardson
    logical :: no_guess                                                 !< disable guessing Eigenfunction from previous level of Richardson
    logical :: rich_conv                                                !< if Richarson extrapolation has converged
    complex(dp), allocatable :: sol_val_rich(:,:,:)                     !< Richardson array of eigenvalues
    real(dp), allocatable :: x_axis_rich(:,:)                           !< x axis for plot of Eigenvalues with Richardson level
    real(dp), allocatable :: max_rel_err(:)                             !< maximum relative error for all Richardson levels
    integer, allocatable :: loc_max_rel_err(:,:)                        !< location of maximum of relative error

contains
    !> Returns string with  possible extension with Richardson  level or nothing
    !! if only one level and one parallel job.
    elemental character(len=max_str_ln) function rich_info()
        if (max_it_rich.gt.1) then
            rich_info = ' for Richardson level '//trim(i2str(rich_lvl))
        else
            rich_info = ''
        end if
    end function rich_info
end module rich_vars
```

D.72 /Users/toonweyens/Code/PB3D/Modules/SLEPC_ops.f90 File Reference

Modules

- module slepc_ops

  Operations that use SLEPC (and PETSC) routines.

Functions/Subroutines

- integer function, public slepc_ops::solve_ev_system_slepc (mds, grid_sol, x, vac, sol)

  Solve the eigenvalue system using SLEPC.
- integer function, public slepc_ops::start_slepc (n_r_sol)

  This subroutine starts PETSC and SLEPC with the correct number of processors.
- integer function, public slepc_ops::setup_mats (mds, grid_sol, x, a, b)

  Sets up the matrices $\overline{A}$ and $\overline{B}$ in the EV problem $\overline{A}\vec{X} = \lambda\overline{B}\vec{X}$..
- integer function, public slepc_ops::set_bc (mds, grid_sol, x, vac, a, b)

  Sets the boundary conditions deep in the plasma (1) and at the edge (2).
- integer function, public slepc_ops::setup_solver (x, a, b, solver)

  Sets up EV solver.
- integer function, public slepc_ops::setup_guess (sol, a, solver)

  Sets up guess in solver.
- integer function, public slepc_ops::get_solution (solver)

  Get the solution vectors.
- integer function, public slepc_ops::summarize_solution (solver, max_n_ev)

  Summarizes solution.
- integer function, public slepc_ops::store_results (mds, grid_sol, sol, solver, max_n_ev, a, b)

  Stores the results.
- integer function, public slepc_ops::stop_slepc (a, b, solver)

  Stop PETSC and SLEPC.

Variables

- logical, public slepc_ops::debug_setup_mats = .false.

  plot debug information for setup_mats
- logical, public slepc_ops::debug_set_bc = .false.

  plot debug information for set_BC
- logical, public slepc_ops::test_diff = .false.

  test introduction of numerical diff

D.73 SLEPC_ops.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Operations that use SLEPC (and PETSC) routines.
!!
!! \note The routines here require a \b TRIMMED solution grid!
!-----------------------------------------------------------------------------!
!> \see
!! References:
!! \cite Hernandez2005slepc
!-----------------------------------------------------------------------------!
module slepc_ops
#include <PB3D_macros.h>
#include <wrappers.h>
! for slepc 3.9.0:
#include <slepc/finclude/slepceps.h>
! for slepc 3.6.0:
!#include <slepc/finclude/slepcepsdef.h>
! for slepc 3.5.3:
!#include <finclude/slepcepsdef.h>
    use str_utilities
    use output_ops
    use messages
    use slepceps
    use num_vars, only: iu, dp, max_str_ln
    use grid_vars, only: grid_type
    use x_vars, only: x_1_type, x_2_type, modes_type
    use vac_vars, only: vac_type
    use sol_vars, only: sol_type

    implicit none
    private
    public solve_ev_system_slepc, start_slepc, setup_mats, set_bc, &
        &setup_solver, setup_guess, get_solution, summarize_solution, &
        &store_results, stop_slepc
#if ldebug
    public debug_setup_mats, debug_set_bc, test_diff
#endif

    ! global variables
    petscreal :: step_size                                          ! step size in flux coordinates
    petscint :: n_r                                                 ! n_r of solution
    petscint :: loc_n_r                                             ! loc_n_r of solution
    integer, allocatable :: c_tot(:,:,:)                            ! total c corresponding to symmetric and asymmetric tensorial variables
    logical :: use_hermitian = .false.                              ! use hermitian matrices (does not work well currently, see http://lists.mcs.anl.gov/pipermail/petsc-users/2016-October/030781.html)
#if ldebug
    logical :: debug_setup_mats = .false.                           !< plot debug information for setup_mats \ldebug
    logical :: debug_set_bc = .false.                               !< plot debug information for set_BC \ldebug
    logical :: test_diff = .false.                                  !< test introduction of numerical diff \ldebug
    real(dp) :: diff_coeff                                          !< diff coefficient \ldebug
#endif

contains
    !> Solve the eigenvalue system using SLEPC.
    !!
    !! This subroutine sets up the matrices A  ad B of the generalized EV system
    !! described in \cite Weyens2017PB3D and  solves them using the SLEPC suite.
    !! The solutions closest to a target indicated by \c EV_guess are obtained.
    !! \see read_input_opts()
    !!
    !! \return ierr
    integer function solve_ev_system_slepc(mds,grid_sol,X,vac,sol) &
        &result(ierr)

        use num_vars, only: matrix_slepc_style
        use rich_vars, only: use_guess
#if ldebug
        use num_vars, only: ltest
        use input_utilities, only: get_real, get_log
#endif

        character(*), parameter :: rout_name = 'solve_EV_system_SLEPC'

        ! input / output
        type(modes_type), intent(in) :: mds                         !< general modes variables
        type(grid_type), intent(in) :: grid_sol                     !< solution grid
        type(x_2_type), intent(in) :: x                             !< field-averaged perturbation variables (so only first index)
        type(vac_type), intent(in) :: vac                           !< vacuum variables
```

```
        type(sol_type), intent(inout) :: sol                                    !< solution variables

        ! local variables
        mat, target :: a                                                        ! matrix A in EV problem A X = lambda B X
        mat, target :: b                                                        ! matrix B in EV problem A X = lambda B X
        eps :: solver                                                           ! EV solver
        petscint :: max_n_ev                                                    ! how many solutions saved
        character(len=max_str_ln) :: err_msg                                    ! error message

        ! initialize ierr
        ierr = 0

        ! initialization
        call writo('Initialize generalized Eigenvalue problem')

        ! start SLEPC
        ierr = start_slepc(grid_sol%n(3))
        chckerr('')

        ! set up step size
        step_size = (grid_sol%r_F(grid_sol%n(3))-grid_sol%r_F(1))/&
            &(grid_sol%n(3)-1)                                                  ! constant

        ! set up the matrix
        call writo('Set up matrices')
        call lvl_ud(1)
        ierr = setup_mats(mds,grid_sol,x,a,b)
        chckerr('')
        call lvl_ud(-1)

#if ldebug
        if (debug_setup_mats) then
            call writo('Testing if A and B are Hermitian by multiplying with &
                &their Hermitian Transpose and subtracting 1')
            call lvl_ud(1)

            ierr = test_mat_hermiticity(a,'A')
            chckerr('')

            ierr = test_mat_hermiticity(b,'B')
            chckerr('')

            call lvl_ud(-1)
        end if
#endif

        ! set boundary conditions
        call writo('Set up boundary conditions')
        call lvl_ud(1)

        select case (matrix_slepc_style)
            case (1)                                                            ! sparse
                ierr = set_bc(mds,grid_sol,x,vac,a,b)
                chckerr('')
#if ldebug
                if (debug_set_bc) then
                    call writo('Testing if AFTER INSERTING BC''s, A and B &
                        &are Hermitian by multiplying with their &
                        &Hermitian Transpose and subtracting 1')
                    call lvl_ud(1)

                    ierr = test_mat_hermiticity(a,'A_BC')
                    chckerr('')

                    ierr = test_mat_hermiticity(b,'B_BC')
                    chckerr('')

                    call lvl_ud(-1)
                end if
#endif
            case (2)                                                            ! shell
                ierr = 1
                err_msg = 'NOT YET IMPLEMENTED FOR SHELL MATRICES'
                chckerr(err_msg)
        end select

        call lvl_ud(-1)

        ! set up solver
        call writo('Set up EV solver with defaults')
        call lvl_ud(1)
#if ldebug
        if (ltest) then
            call writo('Test spectrum of A or B instead of solving &
                &generalized Eigenvalue problem?')
            if (get_log(.false.)) then
                call writo('Spectrum of A (true) or B (false)?')
```

```
                call lvl_ud(1)
                if (get_log(.true.)) then                                   ! A
                    call writo('Testing spectrum of A')
                    ierr = setup_solver(x,a,petsc_null_mat,solver)
                    chckerr('')
                else                                                        ! B
                    call writo('Testing spectrum of B')
                    ierr = setup_solver(x,b,petsc_null_mat,solver)
                    chckerr('')
                end if
                call lvl_ud(-1)
            else
                ierr = setup_solver(x,a,b,solver)
                chckerr('')
            end if
        else
#endif
            ierr = setup_solver(x,a,b,solver)
            chckerr('')
#if ldebug
        end if
#endif
        call lvl_ud(-1)

        ! set up guess
        if (use_guess) then
            call writo('Set up guess')
            call lvl_ud(1)
            ierr = setup_guess(sol,a,solver)
            chckerr('')
            call lvl_ud(-1)
        end if

        ! get solution
        call writo('Get solution')

        ierr = get_solution(solver)
        chckerr('')

        ! summarize solution
        call writo('Summarize solution')

        ierr = summarize_solution(solver,max_n_ev)
        chckerr('')

        ! store results
        call writo('Store results for '//trim(i2str(max_n_ev))//' least &
            &stable Eigenvalues')

        ierr = store_results(mds,grid_sol,sol,solver,max_n_ev,a,b)
        chckerr('')

        ! finalize
        call writo('Finalize SLEPC')

        ! stop SLEPC
        ierr = stop_slepc(a,b,solver)
        chckerr('')
#if ldebug
    contains
        ! Test Hermiticity explicitely.
        !> \private
        integer function test_mat_hermiticity(mat,mat_name) result(ierr)
            use num_vars, only: data_dir
            !use num_vars, only: rank

            character(*), parameter :: rout_name = 'test_mat_hermiticity'

            ! input / output
            mat, intent(in) :: mat                                          ! matrix to test
            character(len=*) :: mat_name                                    ! name of matrix

            ! local variables
            mat :: mat_t                                                    ! Hermitian transpose of mat
            mat :: mat_loc                                                  ! copy of mat, but without block storage
(Hermitian transpose does not work for block)
            petscscalar :: one = 1.0                                        ! one
            petscviewer :: file_viewer                                      ! viewer to write matrix to file
            character(len=max_str_ln) :: file_name                          ! file name

            ! initialize ierr
            ierr = 0

            ! duplicate mat into mat_loc
            call matconvert(mat,matmpiaij,mat_initial_matrix,mat_loc,ierr)
            chckerr('Failed to convert mat into mat_loc')

```

```
            ! test if mat is hermitian
            call mathermitiantranspose(mat_loc,mat_initial_matrix,mat_t,ierr)
            chckerr('Hermitian transpose of mat failed')
            call mataxpy(mat_t,-one,mat_loc,same_nonzero_pattern,ierr)
            chckerr('mat-mat_t failed')

            ! visualize the matrices
            call petscoptionssetvalue(petsc_null_options,'-draw_pause','-1',&
                &ierr)
            chckerr('Failed to set option')
            call writo(trim(mat_name)//' =')
            call matview(mat,petsc_viewer_stdout_world,ierr)
            !call MatView(mat,PETSC_VIEWER_DRAW_WORLD,ierr)
            chckerr('Failed to view')
            call writo(trim(mat_name)//' - '//trim(mat_name)//'^*T =')
            call matview(mat_t,petsc_viewer_stdout_world,ierr)
            chckerr('Failed to view')

            ! destroy matrices
            call matdestroy(mat_t,ierr)
            chckerr('Failed to destroy matrix mat_t')
            call matdestroy(mat_loc,ierr)
            chckerr('Failed to destroy matrix mat_loc')

            !! duplicate mat into mat_loc
            !call MatDuplicate(mat,MAT_SHARE_NONZERO_PATTERN,mat_loc,ierr)
            !CHCKERR('failed to duplicate mat into mat_loc')

            ! write real part to file
            !file_name = trim(data_dir)//'/'//trim(mat_name)//'_RE'
            file_name = trim(data_dir)//'/'//trim(mat_name)
            call petscviewerasciiopen(petsc_comm_world,trim(file_name),&
                &file_viewer,ierr)
            chckerr('Unable to open file viewer')
            !call PetscViewerSetFormat(file_viewer,PETSC_VIEWER_ASCII_DENSE,ierr)
            call petscviewersetformat(file_viewer,petsc_viewer_ascii_matlab,ierr)
            chckerr('Unable to set format')
            !call MatView(mat_loc,file_viewer,ierr)
            call matview(mat,file_viewer,ierr)
            chckerr('Unable to write matrix to file')
            call petscviewerdestroy(file_viewer,ierr)
            chckerr('Unable to destroy file viewer')
            !if (rank.eq.0) then
                !call execute_command_line('tail -n +3 '//trim(file_name)//&
                    !&' > tempfile && mv tempfile '//trim(file_name),&
                    !&EXITSTAT=ierr)
                !CHCKERR('Failed to execute command')
            !end if

            !! write imaginary part to file
            !file_name = trim(data_dir)//'/'//trim(mat_name)//'_IM'
            !call MatCopy(mat,mat_loc,MAT_SHARE_NONZERO_PATTERN,ierr)            ! mat_loc has same structure as mat
            !CHCKERR('Failed to copy mat into mat_loc')
            !call MatImaginaryPart(mat_loc,ierr)
            !CHCKERR('Failed to get imaginary part')
            !call PetscViewerASCIIOpen(PETSC_COMM_WORLD,trim(file_name),&
                !&file_viewer,ierr)
            !CHCKERR('Unable to open file viewer')
            !call PetscViewerSetFormat(file_viewer,PETSC_VIEWER_ASCII_DENSE,ierr)
            !CHCKERR('Unable to set format')
            !call MatView(mat_loc,file_viewer,ierr)
            !CHCKERR('Unable to write matrix to file')
            !call PetscViewerDestroy(file_viewer,ierr)
            !CHCKERR('Unable to destroy file viewer')
            !if (rank.eq.0) then
                !call execute_command_line('tail -n +3 '//trim(file_name)//&
                    !&' > tempfile && mv tempfile '//trim(file_name),&
                    !&EXITSTAT=ierr)
                !CHCKERR('Failed to execute command')
            !end if

            !! destroy matrices
            !call MatDestroy(mat_loc,ierr)
            !CHCKERR('Failed to destroy matrix mat_loc')
        end function test_mat_hermiticity
#endif
    end function solve_ev_system_slepc

    !> This  subroutine  starts PETSC  and  SLEPC  with  the correct  number  of
    !! processors
    !!
    !! \return ierr
    integer function start_slepc(n_r_sol) result(ierr)
        use num_vars, only: n_procs, sol_n_procs
        use files_ops, only: opt_args

        character(*), parameter :: rout_name = 'start_SLEPC'
```

```

        ! input / output
        integer, intent(in) :: n_r_sol                                          !< nr. of normal points in solution grid

        ! local variables
        petscbool :: flg                                                        ! flag to catch options
        petscint :: id                                                          ! counters
        character(len=max_str_ln) :: option_name                                ! options
        character(len=max_str_ln) :: err_msg                                    ! error message

        ! initialize ierr
        ierr = 0

        ! user message
        call writo('initialize SLEPC...')
        call lvl_ud(1)

        ! use MPI_Comm_world for PETSC_COMM_WORLD
        petsc_comm_world = mpi_comm_world
        if (n_procs.gt.n_r_sol) then                                            ! too many processors
            call writo('using too many processors: '//trim(i2str(n_procs))//&
                &', while beyond '//trim(i2str(n_r_sol))//&
                &' does not bring improvement',warning=.true.)
        end if
        call slepcinitialize(petsc_null_character,ierr)                         ! initialize SLEPC
        chckerr('SLEPC failed to initialize')

        ! output
        call writo('slepc started with '//trim(i2str(n_procs))&
            &//' processes')
        if (sol_n_procs.lt.n_procs) call writo('(but only '//&
            &trim(i2str(sol_n_procs))//' processes will be used)')

        call lvl_ud(-1)

        ! checking for complex numbers
        call writo('run tests...')
#if defined(PETSC_USE_COMPLEX)
        ! OK
#else
        err_msg = 'PETSC and SLEPC have to be configured and compiled &
            &with the option "--with-scalar-type=complex"'
        call slepcfinalize(ierr)
        ierr = 1
        chckerr(err_msg)
#endif

        ! catch options so they don't give a Petsc warning
        do id = 1,size(opt_args)
            option_name = opt_args(id)
            if (trim(option_name).ne.'') then
                call petscoptionshasname(petsc_null_options,&
                    &petsc_null_character,trim(option_name),flg,ierr)           ! Petsc 3.7.6
                !call PetscOptionsHasName(PETSC_NULL_CHARACTER,&
                    !&trim(option_name),flg,ierr)                               ! Petsc 3.6.4
                err_msg = 'Failed to decide whether option has a name'
                chckerr(err_msg)
            end if
        end do
    end function start_slepc

    !> Sets      up     the      matrices     \f$\overline{\text{A}}\f$      and
    !! \f$\overline{\text{B}}\f$  in  the  EV  problem  \f$  \overline{\text{A}}
    !! \vec{X} = \lambda \overline{\text{B}} \vec{X}. \f$
    !!
    !! The matrices are set up in the solution grid, so some processes might not
    !! have any  part in the storage  thereof (if \c sol_n_procs  < \c n_procs),
    !! but each process sets  up the part of the grid  that corresponds to their
    !! own normal range in the solution grid.
    !!
    !! \return ierr
    integer function setup_mats(mds,grid_sol,X,A,B) result(ierr)
        use num_vars, only: ndps => norm_disc_prec_sol, matrix_slepc_style, &
            &rank, sol_n_procs, bc_style, norm_disc_style_sol
        use grid_utilities, only: trim_grid
        use slepc_utilities, only: insert_block_mat
        use x_vars, only: n_mod_x
#if ldebug
        use num_vars, only: ltest
        use input_utilities, only: get_real, get_log
#endif

        use num_utilities, only: c

        character(*), parameter :: rout_name = 'setup_mats'

        ! input / output
```

```
        type(modes_type), intent(in) :: mds                                     !< general modes variables
        type(grid_type), intent(in) :: grid_sol                                 !< solution grid
        type(x_2_type), intent(in), target :: x                                 !< field-averaged perturbation variables (so only first index)
        mat, intent(inout) :: a                                                 !< matrix A
        mat, intent(inout) :: b                                                 !< matrix B

        ! local variables
        type(grid_type) :: grid_sol_trim                                        ! trimmed solution grid
        integer :: norm_id(2)                                                   ! untrimmed normal indices for trimmed solution grid
        petscint :: kd, id                                                      ! counter
        petscint :: i_lims(2)                                                   ! lower and upper limit of grid
        petscint, allocatable :: d_nz(:)                                        ! nr. of diagonal non-zeros
        petscint, allocatable :: o_nz(:)                                        ! nr. of off-diagonal non-zeros
        character(len=max_str_ln) :: err_msg                                    ! error message

        ! local variables also used in child routines
        petscint :: bulk_i_lim(2)                                               ! absolute limits of bulk matrix (excluding the BC's)

        ! initialize ierr
        ierr = 0

        ! user output
        select case (norm_disc_style_sol)
            case (1)                                                            ! central finite differences
                call writo('Normal discretization with central finite &
                    &differences of order '//trim(i2str(ndps))//&
                    &', stencil width '//trim(i2str(2*ndps+1)))                 ! ndps left and ndps right
            case (2)                                                            ! left finite differences
                call writo('Normal discretization with left finite &
                    &differences of order '//trim(i2str(ndps))//&
                    &', stencil width '//trim(i2str(ndps+1)))                   ! only ndps left
            case default
                err_msg = 'No solution normal discretization style associated &
                    &with '//trim(i2str(norm_disc_style_sol))
                ierr = 1
                chckerr(err_msg)
        end select

#if ldebug
        if (ltest) then
            call writo('Test introduction of numerical friction?')
            if (get_log(.false.)) then
                test_diff = .true.
                call writo('The friction coefficient will be multiplied with &
                    &the central block, and inserted in the off-diagonal parts')
                call writo('How large should it be?')
                if (get_log(.false.)) diff_coeff = get_real(lim_lo=0._dp)
            end if
        end if
#endif

        ! check nr. of modes
        if (n_mod_x.ne.x%n_mod(1) .or. n_mod_x.ne.x%n_mod(2)) then
            ierr = 1
            err_msg = 'Need square matrix of size [n_mod_X:n_mod_X]'
            chckerr(err_msg)
        end if

        ! setup total c if not yet done
        if (.not.allocated(c_tot)) then
            allocate(c_tot(n_mod_x,n_mod_x,2))
            do kd = 1,n_mod_x
                do id = 1,n_mod_x
                    c_tot(id,kd,1) = c([id,kd],.true.,n_mod_x)
                    c_tot(id,kd,2) = c([id,kd],.false.,n_mod_x)
                end do
            end do
        end if

        ! trim grids
        ierr = trim_grid(grid_sol,grid_sol_trim,norm_id)
        chckerr('')

        ! set up loc_n_r and n_r
        if (rank.lt.sol_n_procs) then
            loc_n_r = grid_sol_trim%loc_n_r
        else
            loc_n_r = 0
        end if
        n_r = grid_sol_trim%n(3)

        ! set up bulk matrix absolute limits
        ierr = set_bulk_lims(grid_sol_trim,bulk_i_lim)
        chckerr('')
```

```
        i_lims = [grid_sol_trim%i_min, grid_sol_trim%i_max]

        ! for BC_style 3 with symmetric finite differences, extend the grid
        if (norm_disc_style_sol.eq.1 .and. bc_style(2).eq.3) then
            n_r = n_r + ndps
            if (rank.eq.sol_n_procs-1) then
                loc_n_r = loc_n_r + ndps
                i_lims(2) = i_lims(2) + ndps
            end if
        end if

        ! setup matrix A and B
        select case (matrix_slepc_style)
            case (1)                                                        ! sparse
                ! calculate nonzeros
                call set_nonzeros(i_lims)

                ! create matrix A
                call matcreatebaij(petsc_comm_world,n_mod_x,n_mod_x*loc_n_r,&
                    &n_mod_x*loc_n_r,n_mod_x*n_r,n_mod_x*n_r,&
                    &petsc_null_integer,d_nz,petsc_null_integer,o_nz,a,ierr)
                err_msg = 'MatCreateAIJ failed for matrix A'
                chckerr(err_msg)

                ! deallocate tot_nz, d_nz and o_nz
                deallocate(d_nz,o_nz)

                ! fill the matrix A
                ierr = fill_mat(x%PV_0(1,:,norm_id(1):norm_id(2),:),&
                    &x%PV_1(1,:,norm_id(1):norm_id(2),:),&
                    &x%PV_2(1,:,norm_id(1):norm_id(2),:),bulk_i_lim,a)
                chckerr('')

                call writo('matrix A set up:')
                ierr = disp_mat_info(a)
                chckerr('')

                ! duplicate A into B
                call matduplicate(a,mat_share_nonzero_pattern,b,ierr)           ! B has same structure as A
                chckerr('failed to duplicate A into B')

                ! fill the matrix B
                ierr = fill_mat(x%KV_0(1,:,norm_id(1):norm_id(2),:),&
                    &x%KV_1(1,:,norm_id(1):norm_id(2),:),&
                    &x%KV_2(1,:,norm_id(1):norm_id(2),:),bulk_i_lim,b)
                chckerr('')

                call writo('matrix B set up:')
                ierr = disp_mat_info(b)
                chckerr('')
            case (2)                                                        ! shell
                ierr = 1
                err_msg = 'NOT YET IMPLEMENTED FOR SHELL MATRICES'
                chckerr(err_msg)
        end select

        ! clean up
        call grid_sol_trim%dealloc()
    contains
        ! Sets the limits of the indices of the bulk matrix, depending on the BC
        ! style.
        !> \private
        integer function set_bulk_lims(grid_sol,i_lim) result(ierr)
            character(*), parameter :: rout_name = 'set_bulk_lims'

            ! input / output
            type(grid_type), intent(in) :: grid_sol                         ! solution grid
            integer, intent(inout) :: i_lim(2)                              ! min and max of bulk limits

            ! initialize ierr
            ierr = 0

            ! set up i_min
            select case (bc_style(1))
                case (1)
                    i_lim(1) = max(grid_sol%i_min,ndps+1)                   ! first ndps rows write left BC's
                case (2:4)
                    err_msg = 'Left BC''s cannot have BC type '//&
                        &trim(i2str(bc_style(1)))
                    ierr = 1
                    chckerr(err_msg)
                case default
                    err_msg = 'No BC style associated with '//&
                        &trim(i2str(bc_style(1)))
                    ierr = 1
                    chckerr(err_msg)
            end select
```

```fortran

            ! set up i_max
            ! Note:   for   BC_style(2)  =   1,   one   could  argue   that   we
            ! should  use  n_r-ndps.  This,  however,  generates  an  error  for
            ! norm_disc_style_sol =  2 where these  elements are not  written in
            ! the main bulk assembly, so that petsc thinks there are no nonzeros
            ! there. This is avoided by using n_r.
            select case (bc_style(2))
                case (1)
                    i_lim(2) = min(grid_sol%i_max,n_r)                          ! last ndps rows write right BC's, will be overwritten
                case (2:4)
                    i_lim(2) = min(grid_sol%i_max,n_r)                          ! right BC is added later
                case default
                    err_msg = 'No BC style associated with '//&
                        &trim(i2str(bc_style(2)))
                    ierr = 1
                    chckerr(err_msg)
            end select
        end function set_bulk_lims

        ! Fills a  matrix according to norm_disc_prec_sol [ADD REF]:
        !   1. first order accuracy for all terms
        !   2. higher order accuracy for internal first order derivatives
        ! It is used  for both the matrix  A and B, corresponding  to the plasma
        ! potential energy and the (perpendicular) kinetic energy.
        !
        ! Makes use of mds, n_r and  grid_sol_trim
        !> \private
        integer function fill_mat(V_0,V_1,V_2,bulk_i_lim,mat) result(ierr)
            use num_utilities, only: con, calc_coeff_fin_diff

            character(*), parameter :: rout_name = 'fill_mat'

            ! input / output
            petscscalar, intent(in) :: v_0(:,:,:)                               ! either PV_int_0 or KV_int_0 in equilibrium normal grid
            petscscalar, intent(in) :: v_1(:,:,:)                               ! either PV_int_1 or KV_int_1 in equilibrium normal grid
            petscscalar, intent(in) :: v_2(:,:,:)                               ! either PV_int_2 or KV_int_2 in equilibrium normal grid
            petscint, intent(in) :: bulk_i_lim(2)                               ! absolute limits of bulk matrix (excluding the BC's)
            mat, intent(inout) :: mat                                           ! either A or B

            ! local variables (not to be used in child routines)
            character(len=max_str_ln) :: err_msg                                ! error message
            petscscalar, allocatable :: loc_block(:,:)                          ! [n_mod_X:n_mod_X] block matrix for 1 normal point
            petscreal, allocatable :: ndc(:)                                    ! normal discretization coefficients
            petscint :: id, jd, kd                                              ! counters
            petscint :: kd_loc                                                  ! kd in local variables
            petscint :: k, m                                                    ! counters
            petscint :: ndc_ind                                                 ! index for finite differences
            petscint :: st_size                                                 ! stencil size
#if ldebug
            petscscalar, allocatable :: loc_block_0_backup(:,:)                 ! backed up V_0 local block
#endif

            ! for tests
            petscint :: r_sol_start, r_sol_end                                  ! start block and end block

            ! initialize ierr
            ierr = 0

            ! test whether the matrix range coincides with i_min and i_max
            call matgetownershiprange(mat,r_sol_start,r_sol_end,ierr)           ! starting and ending row r_sol_start and r_sol_end
            err_msg = 'Couldn''t get ownership range of matrix'
            chckerr(err_msg)
            r_sol_start = r_sol_start/n_mod_x                                   ! count per block
            r_sol_end = r_sol_end/n_mod_x                                       ! count per block
            if (norm_disc_style_sol.eq.1 .and. bc_style(2).eq.3) then
                if (rank.eq.sol_n_procs-1) r_sol_end = r_sol_end - ndps
            end if
            if (rank.lt.sol_n_procs) then
                if (grid_sol_trim%i_min.ne.r_sol_start+1) then
                    ierr = 1
                    err_msg = 'start of matrix in this process does not &
                        &coincide with tabulated value'
                    chckerr(err_msg)
                end if
                if (grid_sol_trim%i_max.ne.r_sol_end) then
                    ierr = 1
                    err_msg = 'end of matrix in this process does not &
                        &coincide with tabulated value'
                    chckerr(err_msg)
```

```
                end if
            end if

            ! allocate local block
            allocate(loc_block(n_mod_x,n_mod_x))

            ! iterate over all rows of this rank
            do kd = bulk_i_lim(1)-1,bulk_i_lim(2)-1                         ! (indices start with 0 here)
                ! set up local kd
                kd_loc = kd+2-grid_sol_trim%i_min

                ! set up ndc
                select case (norm_disc_style_sol)
                    case (1)                                                ! central differences
                        if ((bc_style(2).eq.2 .or. bc_style(2).eq.4) &
                            &.and. kd.gt.n_r-1-ndps) then                   ! asymmetric finite differences: shifted indices
                            ndc_ind = 1+2*ndps - (n_r-1-kd)
                        else
                            ndc_ind = 1+ndps
                        end if
                        st_size = 2*ndps
                    case (2)                                                ! left differences
                        ndc_ind = 1+ndps
                        st_size = ndps
                end select
                ierr = calc_coeff_fin_diff(1,st_size+1,ndc_ind,ndc)
                chckerr('')
                ndc = ndc/step_size                                         ! scale by step size

                ! fill the blocks

                ! -------------!
                ! BLOCKS ~ V_0 !
                ! -------------!
                ! fill local block
                do m = 1,n_mod_x
                    do k = 1,n_mod_x
                        loc_block(k,m) = &
                            &con(sum(v_0(:,kd_loc,c_tot(k,m,1))),&
                            &[k,m],.true.)                                  ! symmetric matrices need con()
                    end do
                end do
                loc_block = loc_block*step_size                             ! include step size of normal integral

#if ldebug
                if (test_diff) then
                    ! back up the V_0 block
                    allocate(loc_block_0_backup(n_mod_x,n_mod_x))
                    loc_block_0_backup = loc_block
                end if
#endif

                ! add block to kd + (0,0)
                ierr = insert_block_mat(mds,loc_block,mat,kd,[0,0],n_r)
                chckerr('')

                ! -------------!
                ! BLOCKS ~ V_1 !
                ! -------------!
                ! fill local block
                do m = 1,n_mod_x
                    do k = 1,n_mod_x
                        loc_block(k,m) = sum(v_1(:,kd_loc,c_tot(k,m,2)))    ! asymetric matrices don't need con()
                    end do
                end do
                loc_block = loc_block*step_size                             ! include step size of normal integral

                ! add block to kd +
                ! (0,1-ndc_ind:1-st_size+jd)  +  Hermitian conjugate
                do jd = 1,1+st_size
                    ierr = insert_block_mat(mds,loc_block*ndc(jd),mat,&
                        &kd,[0,jd-ndc_ind],n_r,transp=.true.)               ! so that jd = ndc_ind falls on [0,0]
                    chckerr('')
                end do

                ! -------------!
                ! BLOCKS ~ V_2 !
                ! -------------!
                ! fill local block
                do m = 1,n_mod_x
                    do k = 1,n_mod_x
                        loc_block(k,m) = &
                            &con(sum(v_2(:,kd_loc,c_tot(k,m,1))),[k,m],.true.)  ! symmetric matrices need con()
                    end do
                end do
                loc_block = loc_block*step_size                             ! include step size of normal integral
```

```fortran

#if ldebug
                if (test_diff) then
                    loc_block = loc_block + diff_coeff*loc_block_0_backup
                    deallocate(loc_block_0_backup)
                end if
#endif

                ! add block to kd +
                ! (1-ndc_ind:1-st_size+id,1-ndc_ind:1-st_size+jd)
                do jd = 1,1+st_size
                    do id = 1,1+st_size
                        ierr = insert_block_mat(mds,loc_block*ndc(id)*ndc(jd),&
                            &mat,kd,[id-ndc_ind,jd-ndc_ind],n_r)                ! so that id,jd = ndc_ind,ndc_ind falls on [0,0]
                        chckerr('')
                    end do
                end do
            end do

            ! assemble the matrix
            call matassemblybegin(mat,mat_final_assembly,ierr)
            chckerr('Coulnd''t begin assembly of matrix')
            call matassemblyend(mat,mat_final_assembly,ierr)
            chckerr('Coulnd''t end assembly of matrix')

            ! Hermitian matrix
            if (use_hermitian) then
                call matsetoption(mat,mat_hermitian,petsc_true,ierr)
                chckerr('Coulnd''t set option Hermitian')
            end if

            ! clean up
            deallocate(loc_block)
        end function fill_mat

        ! Display information about matrix.
        !> \private
        integer function disp_mat_info(mat) result(ierr)
            character(*), parameter :: rout_name = 'disp_mat_info'

            ! input / output
            mat, intent(inout) :: mat                                           ! either A or B

            ! local variables
            real(dp) :: mat_info(mat_info_size)                                 ! information about matrix

            ! initialize ierr
            ierr = 0

            call lvl_ud(1)

            select case (matrix_slepc_style)
                case (1)                                                        ! sparse
                    call matgetinfo(mat,mat_global_sum,mat_info,ierr)
                    chckerr('')
                    call writo('memory usage: '//&
                        &trim(r2strt(mat_info(mat_info_memory)*1.e-6_dp))//&
                        &' MB')
                    call writo('nonzero''s allocated: '//&
                        &trim(r2strt(mat_info(mat_info_nz_allocated))))
                    if (mat_info(mat_info_nz_unneeded).gt.0._dp) then
                        call writo('of which unused: '//&
                            &trim(r2strt(mat_info(mat_info_nz_unneeded))))
                    end if
                case (2)                                                        ! shell
                    call writo('shell matrix')
            end select

            call lvl_ud(-1)
        end function disp_mat_info

        ! Sets nonzero elements d_nz and o_nz.
        ! Makes use of ndps.
        !> \private
        subroutine set_nonzeros(i_lims)
            ! input / output
            petscint, intent(in) :: i_lims(2)                                   ! lower and upper limit of grid

            ! local variables
            petscint, allocatable :: col_lims(:,:)                              ! column limits of nonzeros
            petscint, allocatable :: tot_nz(:)                                  ! nr. of total non-zeros
            petscint :: st_size(2)                                              ! half stencil size left and right

            ! initialize the numbers of non-zeros in diagonal and off-diagonal
            allocate(tot_nz(loc_n_r)); tot_nz = 0
            allocate(d_nz(loc_n_r)); d_nz = 0
```

```
            allocate(o_nz(loc_n_r)); o_nz = 0

            ! initialize helper variable
            allocate(col_lims(loc_n_r,2)); col_lims = 0

            if (rank.lt.sol_n_procs) then
                ! set (half) stencil size
                select case (norm_disc_style_sol)
                    case (1)                                                ! central finite differences
                        st_size = [ndps,ndps]
                    case (2)                                                ! left finite differences
                        st_size = [ndps,0]
                end select

                ! Set column limits of nonzero columns per row
                !   At row  i the largest nonzero  column is the one  due to the
                !   row  at i+st_size(1),  as this  one will  work back  at most
                !   st_size(1)  places.  The  last  column that  this  row  will
                !   contribute to  is st_size(2)  further, which  makes for  i +
                !   sum(st_size). For the lower bound  a similar argument can be
                !   constructed.
                do kd = 1,loc_n_r
                    col_lims(kd,1) = max(kd+i_lims(1)-1-sum(st_size),1)
                    col_lims(kd,2) =  min(kd+i_lims(1)-1+sum(st_size),n_r)
                end do

                ! calculate number of total nonzero entries:
                !   Number of nonzeros columns.
                tot_nz = col_lims(:,2) - col_lims(:,1) + 1

                ! calculate number of nonzero off-diagonal entries
                !   For row  i, this  is equal  to the  number of  elements with
                !   column index greater than  the maximum row index (i_lims(2))
                !   plus the number of elements with column index lower than the
                !   minimum row index (i_lims(1)).
                do kd = 1,loc_n_r
                    o_nz(kd) = max(0,i_lims(1)-col_lims(kd,1)) + &
                        &max(0,col_lims(kd,2)-i_lims(2))
                end do

                ! calculate number of nonzero diagonal entries
                d_nz = tot_nz-o_nz
            else
                d_nz = 0
                o_nz = 0
            end if

            ! clean up
            deallocate(tot_nz)
        end subroutine set_nonzeros
    end function setup_mats

    !> Sets the boundary conditions deep in the plasma (1) and at the edge (2).
    !!
    !! Through  \c  norm_disc_style_sol it  can  be  chosen whether  the  finite
    !! differences are:
    !!  1. central finite differences
    !!  2. left finite differences
    !!
    !! Possibilities for \c BC_style:
    !!  -# Set to zero:
    !!      - An  artificial Eigenvalue  \c EV_BC is  introduced by  setting the
    !!      diagonal  components  of  A  to  EV_BC  and  of  B  to  1,  and  the
    !!      off-diagonal elements to zero.
    !!  -# Minimization of surface energy through asymmetric fin. differences:
    !!      - For symmetric finite differences, the last \c ndps grid points are
    !!      treated asymmetrically in order not go  over the edge and the vacuum
    !!      term is added to the edge element.
    !!      -  For left  differences, this  is already  standard, so  the method
    !!      becomes identical to 2.
    !!  -# Minimization of surface energy through extension of grid:
    !!      - For  symmetric finite differences,  \c ndps extra grid  points are
    !!      introduced after the  edge and the vacuum term is  added to the edge
    !!      element.
    !!      - For left finite differences, the  vacuum term is just added to the
    !!      edge element, so this method becomes idential to 3.
    !!  -# Explicit introduction of the surface energy minimization:
    !!      -  The equation  due  to  the minimization  of  the  vacuum term  is
    !!      introduced explicitely  as an asymmetric finite  difference equation
    !!      in the last row.
    !! -This is done using left finite differences.
    !!
    !! Makes use of n_r.
    !!
    !! \return ierr
    integer function set_bc(mds,grid_sol,X,vac,A,B) result(ierr)
        use num_vars, only: ndps => norm_disc_prec_sol, bc_style, ev_bc, &
```

```
            &norm_disc_style_sol
        use x_vars, only: n_mod_x
        use mpi_utilities, only: get_ser_var, wait_mpi
        use num_utilities, only: con, c
        use slepc_utilities, only: insert_block_mat

        character(*), parameter :: rout_name = 'set_BC'

        ! input / output
        type(modes_type), intent(in) :: mds                                     !< general modes variables
        type(grid_type), intent(in) :: grid_sol                                 !< solution grid
        type(x_2_type), intent(in) :: x                                         !< field-averaged perturbation variables (so
    only first index)
        type(vac_type), intent(in) :: vac                                       !< vacuum variables
        mat, intent(inout) :: a                                                 !< Matrix A from A X = lambda B X
        mat, intent(inout) :: b                                                 !< Matrix B from A X = lambda B X

        ! local variables
        petscint :: kd                                                          ! counter
        petscint :: n_min, n_max                                                ! absolute limits excluding the BC's
        character(len=max_str_ln) :: err_msg                                    ! error message

        ! initialize ierr
        ierr = 0

        call writo('Preparing variables')
        call lvl_ud(1)

        ! check nr. of modes
        if (n_mod_x.ne.x%n_mod(1) .or. n_mod_x.ne.x%n_mod(2)) then
            ierr = 1
            err_msg = 'Need square matrix of size [n_mod_X:n_mod_X]'
            chckerr(err_msg)
        end if

        call lvl_ud(-1)

        ! set up n_min, depending on BC style
        select case (bc_style(1))
            case (1)
                n_min = ndps                                                    ! dirichlet BC requires stencil
            case (2:4)
                err_msg = 'Left BC''s cannot have BC type '//&
                    &trim(i2str(bc_style(1)))
                ierr = 1
                chckerr(err_msg)
            case default
                err_msg = 'No BC style associated with '//&
                    &trim(i2str(bc_style(1)))
                ierr = 1
                chckerr(err_msg)
        end select

        ! set up n_max, depending on BC style
        select case (bc_style(2))
            case (1)
                select case (norm_disc_style_sol)
                    case (1)                                                    ! central differences
                        n_max = ndps                                            ! dirichlet BC requires stencil
                    case (2)                                                    ! left differences
                        n_max = 1                                               ! dirichlet BC only on last point
                end select
            case (2)
                n_max = 1                                                       ! only 1 element carries vacuum contribution
            case (3)
                n_max = 1                                                       ! only 1 element carries vacuum contribution
            case (4)
                n_max = 1                                                       ! only last element carries BC
            case default
                err_msg = 'No BC style associated with '//&
                    &trim(i2str(bc_style(1)))
                ierr = 1
                chckerr(err_msg)
        end select

        call writo('Setting BC deep within plasma')
        call lvl_ud(1)
        call writo('Using artificial eigenvalue EV_BC = '//trim(r2strt(ev_bc)))

        ! iterate over all positions where to set left BC
        do kd = 1,n_min
            if (grid_sol%i_min.le.kd .and. grid_sol%i_max.ge.kd) then           ! decide which process does the setting
                select case (bc_style(1))
                    case (1)
                        ierr = set_bc_1(kd-1,a,b,.false.)                       ! indices start at 0
                        chckerr('')
                    case (2:4)
```

```
                        err_msg = 'Left BC''s cannot have BC type '//&
                            &trim(i2str(bc_style(1)))
                        ierr = 1
                        chckerr(err_msg)
                    case default
                        err_msg = 'No BC style associated with '//&
                            &trim(i2str(bc_style(1)))
                        ierr = 1
                        chckerr(err_msg)
                end select
            end if
        end do

        ! assemble the matrices (cannot mix overwrite and not)
        call matassemblybegin(a,mat_final_assembly,ierr)
        chckerr('Coulnd''t begin assembly of matrix')
        call matassemblybegin(b,mat_final_assembly,ierr)
        chckerr('Coulnd''t begin assembly of matrix')
        call matassemblyend(a,mat_final_assembly,ierr)
        chckerr('Coulnd''t end assembly of matrix')
        call matassemblyend(b,mat_final_assembly,ierr)
        chckerr('Coulnd''t end assembly of matrix')

        call lvl_ud(-1)

        ! wait to get messages consistent
        ierr = wait_mpi()
        chckerr('')

        call writo('Setting BC at plasma edge')
        call lvl_ud(1)

        ! iterate over all positions where to set right BC
        do kd = grid_sol%n(3),grid_sol%n(3)-n_max+1,-1
            if (grid_sol%i_min.le.kd .and. grid_sol%i_max.ge.kd) then           ! decide which process does the setting
                select case (bc_style(2))
                    case (1)
                        ierr = set_bc_1(kd-1,a,b,.true.)                        ! indices start at 0
                        chckerr('')
                    case (2)
                        ierr = set_bc_2(kd-1,a)                                 ! indices start at 0
                        chckerr('')
                    case (3)
                        ierr = set_bc_3(kd-1,a)                                 ! indices start at 0
                        chckerr('')
                    case (4)
                        ierr = set_bc_4(kd-1,kd-grid_sol%i_min+1,x,a,b,&
                            &grid_sol%n(3))                                     ! indices start at 0
                        chckerr('')
                    case default
                        err_msg = 'No BC style associated with '//&
                            &trim(i2str(bc_style(2)))
                        ierr = 1
                        chckerr(err_msg)
                end select
            end if
        end do

        ! assemble the matrices
        call matassemblybegin(a,mat_final_assembly,ierr)
        chckerr('Coulnd''t begin assembly of matrix')
        call matassemblybegin(b,mat_final_assembly,ierr)
        chckerr('Coulnd''t begin assembly of matrix')
        call matassemblyend(a,mat_final_assembly,ierr)
        chckerr('Coulnd''t end assembly of matrix')
        call matassemblyend(b,mat_final_assembly,ierr)
        chckerr('Coulnd''t end assembly of matrix')

        call lvl_ud(-1)
    contains
        ! set BC style 1:
        ! Set to zero
        !   A(ind,ind) = EV_BC, B(ind,ind) = 1,
        !   A(ind+1..ind+p,ind+1..ind+p) = 0, B(ind+1..ind+p,ind+1..ind+p) = 0,
        ! where ind indicates the row where the BC is centered.
        !> \private
        integer function set_bc_1(r_id,A,B,BC_right) result(ierr)
            character(*), parameter :: rout_name = 'set_BC_1'

            ! input / output
            integer, intent(in) :: r_id                                         ! position at which to set BC
            mat, intent(inout) :: a, b                                          ! Matrices A and B from A X = lambda B X
            logical :: bc_right                                                 ! if BC is at right (so there are vacuum terms)

            ! local variables
            integer :: kd                                                       ! counter
```

```
            petscscalar, allocatable :: loc_block(:,:)                          ! [n_mod_X:n_mod_X] block matrix for 1 normal point
            petscint :: pmone                                                   ! plus or minus one
            petscint :: st_size                                                 ! stencil size

            ! initialize ierr
            ierr = 0

            ! user output
            call writo('Boundary style at row '//trim(i2str(r_id+1))//&
                &': Eigenvector set to zero',persistent=.true.)

            ! initialize local blocks
            allocate(loc_block(n_mod_x,n_mod_x))
            loc_block = 0.0_dp
            do kd = 1,n_mod_x
                loc_block(kd,kd) = 1.0_dp
            end do

            ! set up plus minus one
            if (bc_right) then
                pmone = 1
            else
                pmone = -1
            end if

            ! set up stencil size
            select case (norm_disc_style_sol)
                case (1)                                                        ! central differences
                    st_size = 2*ndps
                case (2)                                                        ! left differences
                    st_size = ndps
            end select

            ! set block r_id + (0,0)
            ierr = insert_block_mat(mds,ev_bc*loc_block,a,r_id,[0,0],n_r,&
                &overwrite=.true.,ind_insert=.true.)
            chckerr('')
            ierr = insert_block_mat(mds,loc_block,b,r_id,[0,0],n_r,&
                &overwrite=.true.,ind_insert=.true.)
            chckerr('')

            ! iterate over range 2
            do kd = 1,st_size
                ! set block r_id +/- (0,kd) and Hermitian conjugate
                ierr = insert_block_mat(mds,0*loc_block,a,r_id,[0,-pmone*kd],&
                    &n_r,overwrite=.true.,transp=.true.,ind_insert=.true.)
                chckerr('')
                ierr = insert_block_mat(mds,0*loc_block,b,r_id,[0,-pmone*kd],&
                    &n_r,overwrite=.true.,transp=.true.,ind_insert=.true.)
                chckerr('')
            end do

            ! clean up
            deallocate(loc_block)
        end function set_bc_1

        ! set BC style 2:
        ! Minimization of surface energy through asymmetric fin. differences
        !> \private
        integer function set_bc_2(r_id,A) result(ierr)
            character(*), parameter :: rout_name = 'set_BC_2'

            ! input / output
            integer, intent(in) :: r_id                                         ! position at which to set BC
            mat, intent(inout) :: a                                             ! Matrices A from A X = lambda B X

            ! initialize ierr
            ierr = 0

            ! user output
            call writo('Boundary style at row '//trim(i2str(r_id+1))//&
                &': Minimization of surface energy through asymmetric finite &
                & differences',persistent=.true.)

            ! -------------!
            ! BLOCKS ~ vac !
            ! -------------!
            ! add block to r_id + (0,0)
            ierr = 2
            chckerr('Vacuum has not been implemented yet!')
            ierr = insert_block_mat(mds,vac%res,a,r_id,[0,0],n_r,&
                &ind_insert=.true.)
            chckerr('')
        end function set_bc_2

        ! set BC style 3:
```

```fortran
        ! Minimization of surface energy through extension of grid
        !> \private
        integer function set_bc_3(r_id,A) result(ierr)
            character(*), parameter :: rout_name = 'set_BC_3'

            ! input / output
            integer, intent(in) :: r_id                                     ! position at which to set BC
            mat, intent(inout) :: a                                         ! Matrices A from A X = lambda B X

            ! initialize ierr
            ierr = 0

            ! user output
            call writo('Boundary style at row '//trim(i2str(r_id+1))//&
                &': Minimization of surface energy through extension of grid',&
                &persistent=.true.)

            ! -------------!
            ! BLOCKS ~ vac !
            ! -------------!
            ! add block to r_id + (0,0)
            ierr = 2
            chckerr('Vacuum has not been implemented yet!')
            ierr = insert_block_mat(mds,vac%res,a,r_id,[0,0],n_r,&
                &ind_insert=.true.)
            chckerr('')
        end function set_bc_3

        ! set  BC style 4:
        ! Explicit introduction of the surface energy minimization
        !   V1^T X + V2 X' + delta_vac X = 0 at surface
        !
        ! Makes use of mds.
        !> \private
        integer function set_bc_4(r_id,r_id_loc,X,A,B,n_r) result(ierr)
            use num_vars, only: norm_disc_style_sol
            use num_utilities, only: calc_coeff_fin_diff

            character(*), parameter :: rout_name = 'set_BC_4'

            ! input / output
            integer, intent(in) :: r_id                                     ! global position at which to set BC (starting at 0)
            integer, intent(in) :: r_id_loc                                 ! local index in perturbation tables
            type(x_2_type), intent(in) :: x                                 ! field-averaged perturbation variables (so only first index)
            mat, intent(inout) :: a, b                                      ! Matrices A and B from A X = lambda B X
            integer, intent(in) :: n_r                                      ! number of grid points of solution grid

            ! local variables
            petscint :: jd                                                  ! counter
            petscint :: k, m                                                ! counters
            petscint :: st_size                                             ! stencil size
            petscreal, allocatable :: ndc(:)                                ! normal discretization coefficients
            petscscalar, allocatable :: loc_block(:,:,:)                    ! [n_mod_X:n_mod_X] block matrix for 1 normal point

            ! initialize ierr
            ierr = 0

            ! user output
            call writo('Boundary style at row '//trim(i2str(r_id+1))//&
                &': Explicit introduction of surface energy minimization',&
                &persistent=.true.)

            ! initialize local blocks
            allocate(loc_block(n_mod_x,n_mod_x,2))

            ! set up ndc
            select case (norm_disc_style_sol)
                case (1)                                                    ! central differences
                    st_size = 2*ndps
                case (2)                                                    ! left differences
                    st_size = ndps
            end select
            ierr = calc_coeff_fin_diff(1,st_size+1,st_size+1,ndc)
            chckerr('')
            ndc = ndc/step_size                                             ! scale by step size

            ! ----------------------!
            ! BLOCKS ~ V1^T and vac !
            ! ----------------------!
            ! fill local blocks for A and B
            do m = 1,n_mod_x
                do k = 1,n_mod_x
                    loc_block(k,m,1) = conjg(sum(&
                        &x%PV_1(1,:,r_id_loc,c([m,k],.false.,n_mod_x))))    ! asymetric matrices don't need con()
```

```
                    loc_block(k,m,2) = conjg(sum(&
                        &x%KV_1(1,:,r_id_loc,c([m,k],.false.,n_mod_x))))        ! asymetric matrices don't need con()
                end do
            end do
            loc_block(:,:,1) = loc_block(:,:,1) + vac%res

            ! set block r_id + (0,0)
            ierr = insert_block_mat(mds,loc_block(:,:,1)*ndc(1+st_size),a,r_id,&
                &[0,0],n_r,overwrite=.true.,ind_insert=.true.)
            chckerr('')
            ! set block r_id + (0,0)
            ierr = insert_block_mat(mds,loc_block(:,:,2)*ndc(1+st_size),b,r_id,&
                &[0,0],n_r,overwrite=.true.,ind_insert=.true.)
            chckerr('')

            ! -------------!
            ! BLOCKS ~ V_2 !
            ! -------------!
            ! fill local blocks for A and B
            do m = 1,n_mod_x
                do k = 1,n_mod_x
                    loc_block(k,m,1) = &
                        &con(sum(x%PV_2(1,:,r_id_loc,c([k,m],.true.,n_mod_x))),&
                        &[k,m],.true.)                                          ! symmetric matrices need con()
                    loc_block(k,m,2) = &
                        &con(sum(x%KV_2(1,:,r_id_loc,c([k,m],.true.,n_mod_x))),&
                        &[k,m],.true.)                                          ! symmetric matrices need con()
                end do
            end do

            ! set block r_id + (0,-st_size:0)
            do jd = -st_size,0
                ierr = insert_block_mat(mds,loc_block(:,:,1)*&
                    &ndc(jd+st_size+1)*ndc(1+st_size),a,&
                    &r_id,[0,jd],n_r,overwrite=.true.,&
                    &ind_insert=.true.)
                chckerr('')
                ierr = insert_block_mat(mds,loc_block(:,:,2)*&
                    &ndc(jd+st_size+1)*ndc(1+st_size),b,&
                    &r_id,[0,jd],n_r,overwrite=.true.,&
                    &ind_insert=.true.)
                chckerr('')
            end do
        end function set_bc_4
    end function set_bc

    !> Sets up EV solver.
    !!
    !! \return ierr
    integer function setup_solver(X,A,B,solver) result(ierr)
        use num_vars, only: n_sol_requested, tol_slepc, max_it_slepc, &
            &ev_guess, solver_slepc_style
        use rich_vars, only: rich_lvl
        use x_vars, only: n_mod_x

        character(*), parameter :: rout_name = 'setup_solver'

        ! input / output
        type(x_2_type), intent(in) :: x                                         !< field-averaged perturbation variables (so only first index)
        mat, intent(in) :: a, b                                                 !< matrix A and B
        eps, intent(inout) :: solver                                            !< EV solver

        ! local variables
        petscscalar :: one = 1.0                                                ! one
        st :: solver_st                                                         ! spectral transformation associated to the solver
        petscint :: n_sol                                                       ! how many solutions can be requested (normally n_sol_requested)
        character(len=max_str_ln) :: err_msg                                    ! error message

        ! initialize ierr
        ierr = 0

        ! check nr. of modes
        if (n_mod_x.ne.x%n_mod(1) .or. n_mod_x.ne.x%n_mod(2)) then
            ierr = 1
            err_msg = 'Need square matrix of size [n_mod_X:n_mod_X]'
            chckerr(err_msg)
        end if

        !call PetscOptionsSetValue(PETSC_NULL_OPTION,'-eps_view','-1',ierr)
        call epscreate(petsc_comm_world,solver,ierr)
        chckerr('EPSCreate failed')

        call epssetoperators(solver,a,b,ierr)                                   ! generalized EV problem A X = lambda B X
        chckerr('EPSetOperators failed')
```

```fortran

        if (use_hermitian) then
            call writo('Set problem type to Generalized Hermitian')
            call epssetproblemtype(solver,eps_ghep,ierr)
            chckerr('Set problem type failed')
        end if

        ! set some options depending on which default style
        select case (solver_slepc_style)
            case (1)                                                        ! Krylov-Schur with shift-invert
                ! Krylov-Schur is the most robust
                call writo('set algorithm to Krylov-Schur')
                call epssettype(solver,epskrylovschur,ierr)
                err_msg = 'Failed to set type to Krylov-Schur'
                chckerr(err_msg)

                ! get spectral transformation
                call epsgetst(solver,solver_st,ierr)
                err_msg = 'Failed to get ST from solver'
                chckerr(err_msg)

                ! shift-invert is  the best method for  generalized EV problems,
                ! because a matrix needs to be inverted anyway
                call writo('use shift-invert')
                call stsettype(solver_st,stsinvert,ierr)
                err_msg = 'Failed to set type to STSINVERT'
                chckerr(err_msg)
            case (2)                                                        ! generalized Davidson with preconditioner
                ! GD might be faster than Krylov-Schur
                call writo('set algorithm to Generalized Davidson')
                call epssettype(solver,epsgd,ierr)
                err_msg = 'Failed to set type to Generalized Davidson'
                chckerr(err_msg)

                ! warning
                call lvl_ud(1)
                call writo('This is often less robust than the Krylov-Schur &
                    &method.',alert=.true.)
                call lvl_ud(1)
                call writo('If it converges to a strange result, try using &
                    &that')
                call lvl_ud(-1)
                call lvl_ud(-1)

                ! get spectral transformation
                call epsgetst(solver,solver_st,ierr)
                err_msg = 'Failed to get ST from solver'
                chckerr(err_msg)

                ! GD works only with a preconditioner
                call writo('Use preconditioner')
                call stsettype(solver_st,stprecond,ierr)
                err_msg = 'Failed to set type to STPRECOND'
                chckerr(err_msg)
            case default
                call writo('SLEPC solver style '//&
                    &trim(i2str(solver_slepc_style))//' not recognized',&
                    &alert=.true.)
        end select

        ! set the guess as target
        call writo('set the target eigenvalue to '//trim(r2str(ev_guess)))
        call epssettarget(solver,ev_guess*one,ierr)
        err_msg = 'Failed to set the target of solver'
        chckerr(err_msg)

        ! request n_sol_requested Eigenpairs
        if (n_sol_requested.gt.n_r*n_mod_x) then
            call writo('max. nr. of solutions requested capped to problem &
                &dimension ('//trim(i2str(n_r*n_mod_x))//')',&
                &warning=.true.)
            call writo('Increase either n_r or number of pol. modes or &
                &decrease n_sol_requested')
            n_sol = n_r*n_mod_x
        else
            n_sol = n_sol_requested
        end if
        call epssetdimensions(solver,n_sol,petsc_decide,&
            &petsc_decide,ierr)
        chckerr('EPSSetDimensions failed')
        call epssettolerances(solver,tol_slepc(rich_lvl),max_it_slepc,ierr)
        chckerr('EPSSetTolerances failed')

        ! user output
        call writo('set tolerance to '//trim(r2str(tol_slepc(rich_lvl))))
        call writo('set maximum nr. of iterations to '//&
            &trim(i2str(max_it_slepc)))
```

```fortran
    end function setup_solver

    !> Sets up guess in solver.
    !!
    !! \return ierr
    integer function setup_guess(sol,A,solver) result(ierr)
        character(*), parameter :: rout_name = 'setup_guess'

        ! input / output
        type(sol_type), intent(in) :: sol                                   !< solution variables
        mat, intent(in) :: a                                                !< matrix A (or B)
        eps, intent(inout) :: solver                                        !< EV solver

        ! local variables
        vec, allocatable :: guess_vec(:)                                    ! guess to solution EV parallel vector
        petscint :: kd                                                      ! counter
        petscint :: n_ev_prev                                               ! nr. of previous solutions
        petscscalar, pointer ::  guess_vec_ptr(:)                           ! pointer to local guess_vec
        character(len=max_str_ln) :: err_msg                                ! error message

        ! initialize ierr
        ierr = 0

        call lvl_ud(1)

        ! set guess for EV if sol vec is allocated
        if (allocated(sol%val)) then
            ! set n_EV_prev
            n_ev_prev = size(sol%val)

            if (n_ev_prev.gt.0) then
                ! user output
                call writo('Set '//trim(i2str(n_ev_prev))//&
                    &' vector(s) as guess')

                ! allocate guess vectors
                allocate(guess_vec(n_ev_prev))

                ! create vecctor guess_vec and set values
                do kd = 1,n_ev_prev
                    ! create the vectors
                    call matcreatevecs(a,guess_vec(kd),petsc_null_vec,ierr)     ! get compatible parallel vectors to matrix A

                    chckerr('Failed to create vector')

                    ! get pointer
                    call vecgetarrayf90(guess_vec(kd),guess_vec_ptr,ierr)
                    chckerr('Failed to get pointer')

                    ! copy the values
                    guess_vec_ptr(1:size(sol%vec(:,:,kd))) = &
                        &reshape([sol%vec(:,:,kd)],&
                        &[size(sol%vec(:,:,kd))])
                    guess_vec_ptr(size(sol%vec(:,:,kd))+1:size(guess_vec_ptr)) &
                        &= guess_vec_ptr(size(sol%vec(:,:,kd)))                 ! some BC's have a grid extension

                    ! return pointer
                    call vecrestorearrayf90(guess_vec(kd),guess_vec_ptr,ierr)
                    chckerr('Failed to restore pointer')

                    !! visualize guess
                    !call VecView(guess_vec(kd),PETSC_VIEWER_STDOUT_WORLD,ierr)
                    !CHCKERR('Cannot view vector')
                end do

                ! set guess
                call epssetinitialspace(solver,n_ev_prev,guess_vec,ierr)
                chckerr('Failed to set guess')

                ! destroy guess vector
                do kd = 1,size(guess_vec)
                    call vecdestroy(guess_vec(kd),ierr)                     ! destroy guess vector
                    err_msg = 'Failed to destroy guess vector '//trim(i2str(kd))
                    chckerr(err_msg)
                end do
                deallocate(guess_vec)
            else
                call writo('No vectors saved to use as guess')
            end if
        end if

        call lvl_ud(-1)
    end function setup_guess

    !> Get the solution vectors.
    !!
    !! \return ierr
```

```
    integer function get_solution(solver) result(ierr)
        character(*), parameter :: rout_name = 'get_solution'

        ! input / output
        eps, intent(inout) :: solver                                        !< EV solver

        ! local variables
        character(len=max_str_ln) :: err_msg                                ! error message

        ! initialize ierr
        ierr = 0

        call lvl_ud(1)

        ! set run-time options
        call epssetfromoptions(solver,ierr)
        chckerr('EPSSetFromOptions failed')

        ! solve EV problem
        call epssolve(solver,ierr)
        err_msg = 'EPS couldn''t find a solution. Maybe you should increase &
            &the number of parallel points.'
        chckerr(err_msg)

        call lvl_ud(-1)
    end function get_solution

    !> Summarizes solution.
    !!
    !! \return ierr
    integer function summarize_solution(solver,max_n_EV) result (ierr)
        use num_vars, only: n_sol_requested

        character(*), parameter :: rout_name = 'summarize_solution'

        ! input / output
        eps, intent(inout) :: solver                                        !< EV solver
        petscint, intent(inout) :: max_n_ev                                 !< nr. of EV's saved, up to \c n_conv

        ! local variables
        petscint :: n_it                                                    ! nr. of iterations
        petscint :: n_conv                                                  ! nr. of converged solutions
        petscint :: n_ev, ncv, mpd                                          ! nr. of requested EV, max. dim of subspace
and for projected problem
        epstype :: eps_type                                                 ! string with name of type of EPS solver
        petscreal :: tol                                                    ! tolerance of EPS solver
        petscint ::  max_it                                                 ! maximum number of iterations

        ! initialize ierr
        ierr = 0

        call lvl_ud(1)

        ! user output
        call epsgettype(solver,eps_type,ierr)                               ! EPS solver type
        chckerr('EPSGetType failed')
        call epsgettolerances(solver,tol,max_it,ierr)                       ! tolerance and max. nr. of iterations
        chckerr('EPSGetTolerances failed')
        call writo(trim(eps_type)//' solver with tolerance '//&
            &trim(r2strt(tol))//' and maximum '//trim(i2str(max_it))//&
            &' iterations')
        call epsgetconverged(solver,n_conv,ierr)                            ! nr. of converged solutions
        chckerr('EPSGetConverged failed')
        call epsgetiterationnumber(solver,n_it,ierr)                        ! nr. of iterations
        chckerr('EPSGetIterationNumber failed')
        call epsgetdimensions(solver,n_ev,ncv,mpd,ierr)                     ! nr. of requested EV, ...
        chckerr('EPSGetDimensions failed')
        call writo('number of iterations: '//trim(i2str(n_it)))
        call writo('number of converged solutions: '//trim(i2str(n_conv)))
        call writo('number of requested solutions: '//trim(i2str(n_ev)))
        call writo('maximum dimension of the subspace to be used by solver: '//&
            &trim(i2str(ncv)))
        call writo('maximum dimension allowed for projected problem : '//&
            &trim(i2str(mpd)))

        ! set maximum nr of solutions to be saved
        if (n_sol_requested.gt.n_conv) then
            if (n_conv.eq.0) then
                call writo('no solutions found',warning=.true.)
            else
                call writo('max. nr. of solutions found only '//&
                    &trim(i2str(n_conv)),warning=.true.)
            end if
            max_n_ev = n_conv
        else
            max_n_ev = n_sol_requested
        end if
```

```fortran

        ! check whether solutions found
        if (max_n_ev.le.0) then
            ierr = 1
            chckerr('No solutions found')
        end if

        call lvl_ud(-1)
    end function summarize_solution

    !> Stores the results.
    !!
    !! \return ierr
    integer function store_results(mds,grid_sol,sol,solver,max_n_EV,A,B) &
        &result(ierr)

        use eq_vars, only: t_0
        use x_vars, only: n_mod_x
        use num_vars, only: use_normalization, ev_bc, prog_name, output_name, &
            &n_procs, rank, tol_slepc, eq_style, output_ev_i, retain_all_sol
        use mpi_utilities, only: get_ser_var
        use rich_vars, only: rich_lvl
        use grid_utilities, only: trim_grid

        character(*), parameter :: rout_name = 'store_results'

        ! input / output
        type(modes_type), intent(in) :: mds                                     !< general modes variables
        type(grid_type), intent(in) :: grid_sol                                 !< solution grid
        type(sol_type), intent(inout) :: sol                                    !< solution variables
        eps, intent(inout) :: solver                                            !< EV solver
        petscint, intent(inout) :: max_n_ev                                     !< nr. of EV's saved, up to \c n_conv
        mat, intent(inout) :: a                                                 !< Matrix A from A X = lambda B X
        mat, intent(inout) :: b                                                 !< Matrix B from A X = lambda B X

        ! local variables
        type(grid_type) :: grid_sol_trim                                        ! trimmed solution grid
        mat :: err_mat                                                          ! A - omega^2 B
        vec :: sol_vec                                                          ! solution EV parallel vector
        vec :: err_vec                                                          ! AX - omega^2 BX
        vec :: e_vec(2)                                                         ! AX and BX
        petscint :: id, jd                                                      ! counters
        petscint :: id_tot                                                      ! counters
        petscint :: one = 1                                                     ! one
        petscint, allocatable :: sol_vec_max_loc(:)                             ! location of sol_vec_max of this process
        petscreal :: error                                                      ! error of EPS solver
        petscreal :: error_est                                                  ! error estimate of EPS solver
        petscreal :: err_norm                                                   ! norm of error
        petscreal :: tol_complex                                                ! tolerance for an EV to be considered complex
        petscreal :: tol_ev_bc = 1.e-3_dp                                       ! tolerance for an EV to be considered due to the BC
        petscreal, parameter :: infinity = 1.e40_dp                             ! beyond this value, modes are not saved
        petscscalar :: sol_val_loc                                              ! local X val
        petscscalar :: err_val                                                  ! X*(A-omega^2B)X
        petscscalar :: e_val(2)                                                 ! X*AX and X*BX
        petscscalar, allocatable :: sol_vec_loc(:,:)                            ! local solution vector, possibly in extended grid
        petscscalar, allocatable :: sol_vec_max(:)                              ! max of sol_vec of all processes, including phase at max
        character(len=max_str_ln) :: full_output_name                           ! full name
        character(len=max_str_ln) :: format_val                                 ! format
        character(len=max_str_ln) :: format_head                                ! header
        character(len=max_str_ln) :: ev_err_str                                 ! String with information about EV error
        character(len=max_str_ln) :: err_msg                                    ! error message
        character(len=2*max_str_ln) :: temp_output_str                          ! temporary output string
        integer :: n_digits                                                     ! nr. of digits for the integer number
        integer :: n_err(3)                                                     ! how many errors there were

        ! initialize ierr
        ierr = 0

        call lvl_ud(1)

        ! trim grid
        ierr = trim_grid(grid_sol,grid_sol_trim)
        chckerr('')

        ! create solution variables
        if (allocated(sol%val)) call sol%dealloc()
        call sol%init(mds,grid_sol_trim,max_n_ev)

        ! create solution vector
        call veccreatempiwitharray(petsc_comm_world,one,loc_n_r*n_mod_x,&
            &n_r*n_mod_x,petsc_null_scalar,sol_vec,ierr)
        chckerr('Failed to create MPI vector with arrays')

```

```
        ! set up EV error string and format string:
        !   1: index of EV
        !   2: Real part
        !   3: Imaginary part
        !   4: relative error ||Ax - EV Bx||_2/||EV x||_2 [1]
        n_err = 0
        ev_err_str = ''
        if (rank.eq.0) then
            n_digits = min(ceiling(log10(1._dp*max_n_ev)),16)
            format_val = '(A2,I'//trim(i2str(n_digits))//',' ",ES23.16," ",&
                &ES23.16," ",ES23.16)'
            format_head = '(A'//trim(i2str(n_digits+3))//',' ",A23," ",A23," ",&
                &A23)'
        end if

        ! open output file for the log
        if (rank.eq.0) then
            full_output_name = prog_name//'_'//trim(output_name)//'_EV_R_'//&
                &trim(i2str(rich_lvl))//'.txt'
            open(output_ev_i,file=full_output_name,status='replace',&
                &iostat=ierr)
            chckerr('Cannot open EV output file')
        end if

        ! output to file
        if (rank.eq.0) then
            if (use_normalization) then
                write(unit=output_ev_i,fmt='(A)',iostat=ierr) &
                    &'# Eigenvalues normalized to the squared Alfven &
                    &frequency omega_A^2 = '
                chckerr('Failed to write')
                select case (eq_style)
                    case (1)                                            ! VMEC
                        write(unit=output_ev_i,fmt='(A)',iostat=ierr) &
                            &'#     ('//trim(r2str(1._dp/t_0))//' Hz)^2 = '//&
                            &trim(r2str(1._dp/t_0**2))//' Hz^2'
                        chckerr('Failed to write')
                    case (2)                                            ! HELENA
                        write(unit=output_ev_i,fmt='(A)',iostat=ierr) &
                            &'#     (MISHKA normalization)'
                        chckerr('Failed to write')
                end select
            else
                write(unit=output_ev_i,fmt='(A)',iostat=ierr) '# Eigenvalues'
                chckerr('Failed to write')
            end if
            write(temp_output_str,format_head) &
                &'#  I                            ', &
                &'real part               ', 'imaginary part          ', &
                &'relative precision      '
            write(unit=output_ev_i,fmt='(A)',iostat=ierr) trim(temp_output_str)
            chckerr('Failed to write')
        end if

        ! initialize helper variables
        allocate(sol_vec_max(n_procs))
        allocate(sol_vec_max_loc(2))
        allocate(sol_vec_loc(n_mod_x,loc_n_r))

        ! start id
        id = 1
        id_tot = 1

        ! store them
        do while (id.le.max_n_ev)
            call vecplacearray(sol_vec,sol_vec_loc,ierr)                ! place array local sol_vec in solution vec
            chckerr('Failed to place array')
            call epsgeteigenpair(solver,id_tot-1,sol%val(id),petsc_null_scalar,&
                &sol_vec,petsc_null_vec,ierr)                           ! get solution EV vector and value (starts at index 0)
            chckerr('EPSGetEigenpair failed')
            sol%vec(:,:,id) = sol_vec_loc(:,1:size(sol%vec,2))
            call epscomputeerror(solver,id_tot-1,eps_error_relative,error,ierr) ! get error (starts at index 0) (petsc 3.6.1)
            !call EPSComputeRelativeError(solver,id_tot-1,error,ierr)            ! get error (starts at index 0) (petsc 3.5.3)
            chckerr('EPSComputeError failed')

            ! set up local solution val
            sol_val_loc = sol%val(id)

            ! tests
            tol_complex = 10*tol_slepc(rich_lvl)
            if (abs(ip(sol%val(id))/rp(sol%val(id))).gt.tol_complex) then       ! test for unphysical complex solution
                ev_err_str = '# WARNING: Unphysical complex Eigenvalue!'
                n_err(1) = n_err(1)+1
                call remove_ev(id,max_n_ev)
```

```
            else if (abs(rp(sol%val(id)-ev_bc)/ev_bc).lt.tol_ev_bc) then        ! test for artificial EV due to BC
                ev_err_str = '# WARNING: Eigenvalue probably due to BC''s &
                    &(artifically set to '//trim(r2strt(ev_bc))//')'
                n_err(2) = n_err(2)+1
                call remove_ev(id,max_n_ev)
            else if (abs(sol%val(id)).gt.infinity) then                         ! test for infinity
                ev_err_str = '# WARNING: The next Eigenvalues are larger &
                    &than '//trim(r2strt(infinity))//' and are discarded'
                n_err(3) = 1
                call remove_ev(id,max_n_ev,remove_next=.true.)
                exit
            end if

            if (rank.eq.0) then
                ! output EV to file
                if (ev_err_str.ne.'') then
                    write(temp_output_str,format_val) '#', id,rp(sol_val_loc),&
                        &ip(sol_val_loc),error                                  ! if error, comment the next line
                else
                    write(temp_output_str,format_val) '', id,rp(sol_val_loc),&
                        &ip(sol_val_loc),error
                end if
                write(unit=output_ev_i,fmt='(A)',iostat=ierr) &
                    &trim(temp_output_str)
                chckerr('Failed to write')

                ! if error, print explanation
                if (ev_err_str.ne.'') then
                    write(unit=output_ev_i,fmt='(A)',iostat=ierr) &
                        &trim(ev_err_str)
                    chckerr('Failed to write')
                end if
            end if

            ! normalize Eigenvectors to make output more easily comparable
            if (ev_err_str.eq.'') then
                ! find local maximum
                sol_vec_max_loc = maxloc(abs(sol%vec(:,:,id)))
                ierr = get_ser_var(&
                    &[sol%vec(sol_vec_max_loc(1),sol_vec_max_loc(2),id)],&
                    &sol_vec_max,scatter=.true.)
                chckerr('')
                ! find global maximum
                sol_vec_max_loc(1) = maxloc(abs(sol_vec_max),1)
                sol%vec(:,:,id) = sol%vec(:,:,id) / &
                    &sol_vec_max(sol_vec_max_loc(1))

                call epsgeterrorestimate(solver,id_tot-1,error_est,ierr)        ! get error estimate
                chckerr('EPSGetErrorEstimate failed')
                ! user message
                call writo('Checking whether A x - omega^2 B x = 0 for EV '//&
                    &trim(i2str(id))//': '//trim(c2strt(sol%val(id))))
                call lvl_ud(1)

                ! set up error matrix A - omega^2 B
                call matduplicate(a,mat_share_nonzero_pattern,err_mat,ierr)
                err_msg = 'failed to duplicate mat into err_mat'
                chckerr(err_msg)
                call matcopy(a,err_mat,mat_share_nonzero_pattern,ierr)          ! err_mat has same structure as A
                chckerr('Failed to copy mat into mat_loc')
                call mataxpy(err_mat,-sol%val(id),b,different_nonzero_pattern,&
                    &ierr)                                                      ! for some reason, SAME_NONZERO_PATTERN does not work
                chckerr('Failed to perform AXPY')

                ! set up error vector
                call vecduplicate(sol_vec,err_vec,ierr)
                chckerr('Failed to duplicate vector')

                ! calculate Ax-lambdaBx
                call matmult(err_mat,sol_vec,err_vec,ierr)
                chckerr('Failed to multiply')

                ! multiply with X* and give output
                call vecdot(err_vec,sol_vec,err_val,ierr)
                chckerr('Failed to do dot product')
                call writo('X*(A-omega^2B)X = '//trim(c2strt(err_val)))

                ! calculate absolute norm
                call vecnorm(err_vec,norm_2,err_norm,ierr)
                chckerr('Failed to calculate norm')

                ! get relative norm
                err_norm = err_norm/abs(sol%val(id))

                ! visualize solution and error
                call writo('error: '//trim(r2str(err_norm))//', given: '//&
```

```
                    &trim(r2str(error))//', estimate: '//trim(r2str(error_est)))
                !!call VecView(err_vec,PETSC_VIEWER_STDOUT_WORLD,ierr)
                !!CHCKERR('Cannot view vector')

                ! set up Energy vector
                call vecduplicate(sol_vec,e_vec(1),ierr)
                chckerr('Failed to duplicate vector')
                call vecduplicate(sol_vec,e_vec(2),ierr)
                chckerr('Failed to duplicate vector')

                ! calculate Ax and Bx
                call matmult(a,sol_vec,e_vec(1),ierr)
                chckerr('Failed to multiply')
                call matmult(b,sol_vec,e_vec(2),ierr)
                chckerr('Failed to multiply')

                ! multiply with X* and give output
                call vecdot(e_vec(1),sol_vec,e_val(1),ierr)
                chckerr('Failed to do dot product')
                call writo('step_size = '//trim(r2str(step_size)))
                call writo('E_pot = X*AX*step_size = '//&
                    &trim(c2str(e_val(1)*step_size)))
                call vecdot(e_vec(2),sol_vec,e_val(2),ierr)
                chckerr('Failed to do dot product')
                call writo('E_kin = X*BX*step_size = '//&
                    &trim(c2str(e_val(2)*step_size)))
                call writo('X*AX/X*BX = '//trim(c2str(e_val(1)/e_val(2))))
                call writo('omega^2   = '//trim(c2str(sol%val(id))))

                ! if stable EV, give warning
                if (rp(sol%val(id)).gt.0._dp) then
                    call writo('The eigenvalue is positive, so the mode could &
                        &be stable.',warning=.true.)
                    call lvl_ud(1)
                    call writo('But this could also be due to value of "ncv" &
                        &that is too low.')
                    call writo('Have a look at the POST output and/or try &
                        &increasing "ncv".')
                    call lvl_ud(-1)
                end if

                ! increment counter
                id = id + 1

                call lvl_ud(-1)

                ! clean up
                call vecdestroy(err_vec,ierr)                                   ! destroy error vector
                chckerr('Failed to destroy err_vec')
                call matdestroy(err_mat,ierr)                                   ! destroy error matrix
                chckerr('Failed to destroy err_mat')
                do jd = 1,size(e_vec)
                    call vecdestroy(e_vec(jd),ierr)                             ! destroy energy vector
                    err_msg = 'Failed to destroy E_vec '//trim(i2str(jd))
                    chckerr(err_msg)
                end do
            else
                ! reinitialize error string if error
                ev_err_str = ''
            end if

            ! reset vector
            call vecresetarray(sol_vec,ierr)
            chckerr('Failed to reset array')

            ! increment total id
            id_tot = id_tot+1
        end do

        ! deallocate helper variables
        deallocate(sol_vec_max_loc)
        deallocate(sol_vec_max)

        ! close output file if master
        if (rank.eq.0) close(output_ev_i)

        ! user output
        if (rank.eq.0) call writo(trim(i2str(max_n_ev))//&
            &' Eigenvalues were written in the file "'//&
            &trim(full_output_name)//'"')
        if (sum(n_err).ne.0) then
            call writo('there were some Eigenvalues that were removed:')
            call lvl_ud(1)

            if (n_err(1).gt.0) call writo(trim(i2str(n_err(1)))//&
                &' unphysical complex Eigenvalues')
            if (n_err(2).gt.0) call writo(trim(i2str(n_err(2)))//&
```

```
                &' Eigenvalues due to the boundary conditions')
            if (n_err(3).gt.0) call writo('The final Eigenvalues became &
                &infinite')

            call lvl_ud(-1)
            if (retain_all_sol) then
                call writo('(But this was overriden using "retain_all_sol")')
            else
                call writo('(Override this behavior using "retain_all_sol")')
            end if
        end if

        if (max_n_ev.gt.0) then
            call writo('basic statistics:')
            call lvl_ud(1)

            call writo('min: '//trim(c2strt(sol%val(1))))
            call writo('max: '//trim(c2strt(sol%val(max_n_ev))))

            call lvl_ud(-1)
        end if

        !!call EPSErrorView(solver,EPS_ERROR_RELATIVE,PETSC_NULL_VIEWER,ierr)
        !!CHCKERR('Failed to print solution')

        call vecdestroy(sol_vec,ierr)                                   ! destroy solution vector
        chckerr('Failed to destroy sol_vec')

        call lvl_ud(-1)

        ! clean up
        call grid_sol_trim%dealloc()
    contains
        ! Removes id'th  Eigenvalue and  -vector. Optionally, all  the following
        ! can be removed as well.
        !> \private
        subroutine remove_ev(id,max_id,remove_next)
            ! input / output
            petscint, intent(inout) :: id                               ! id of faulty values
            petscint, intent(inout) :: max_id                           ! maximum id
            petscbool, intent(in), optional :: remove_next              ! whether all next values have to be removed as well

            ! local variables
            petscscalar, allocatable :: sol_val_loc(:)                  ! local copy of sol_val
            petscscalar, allocatable :: sol_vec_loc(:,:,:)              ! local copy of sol_vec
            petscbool :: remove_next_loc = .false.                      ! local copy of remove_next

            ! only remove if faulty solutions are not optionally retained
            if (retain_all_sol) then
                id = id+1                                               ! increment the solution
            else
                ! set up local remove_next
                if (present(remove_next)) remove_next_loc = remove_next

                ! save old arrays
                allocate(sol_val_loc(max_id))
                allocate(sol_vec_loc(n_mod_x,grid_sol_trim%loc_n_r,max_id))
                sol_val_loc = sol%val
                sol_vec_loc = sol%vec

                ! reallocate arrays
                deallocate(sol%val,sol%vec)
                if (remove_next_loc) then
                    allocate(sol%val(id-1))
                    allocate(sol%vec(n_mod_x,grid_sol_trim%loc_n_r,id-1))
                else
                    allocate(sol%val(max_id-1))
                    allocate(sol%vec(n_mod_x,grid_sol_trim%loc_n_r,max_id-1))
                end if

                ! copy other values
                sol%val(1:id-1) = sol_val_loc(1:id-1)
                sol%vec(:,:,1:id-1) = sol_vec_loc(:,:,1:id-1)
                if (.not.remove_next_loc) then
                    sol%val(id:max_id-1) = sol_val_loc(id+1:max_id)
                    sol%vec(:,:,id:max_id-1) = sol_vec_loc(:,:,id+1:max_id)
                end if

                ! adapt max_id
                if (remove_next_loc) then
                    max_id = id-1
                else
                    max_id = max_id-1
                end if

                ! clean up
```

```
                deallocate(sol_val_loc,sol_vec_loc)
            end if
        end subroutine remove_ev
    end function store_results

    !> Stop PETSC and SLEPC.
    !!
    !! \return ierr
    integer function stop_slepc(A,B,solver) result(ierr)
        character(*), parameter :: rout_name = 'stop_SLEPC'

        ! input / output
        mat, intent(in) :: a, b                                          !< matrices A and B in EV problem
        eps, intent(in) :: solver                                        !< EV solver

        ! initialize ierr
        ierr = 0

        call lvl_ud(1)

        ! destroy objects
        call epsdestroy(solver,ierr)
        chckerr('Failed to destroy EPS solver')
        call matdestroy(a,ierr)
        chckerr('Failed to destroy matrix A')
        call matdestroy(b,ierr)
        chckerr('Failed to destroy matrix B')

        ! stop SLEPC
        call slepcfinalize(ierr)
        chckerr('Failed to Finalize SLEPC')

        call lvl_ud(-1)
    end function stop_slepc
end module
```

D.74 /Users/toonweyens/Code/PB3D/Modules/SLEPC_utilities.f90 File Reference

Modules

- module slepc_utilities

  Numerical utilities related to SLEPC (and PETSC) operations.

Functions/Subroutines

- integer function, public slepc_utilities::insert_block_mat (mds, block, mat, r_id, ind, n_r, transp, overwrite, ind_insert)

  Insert a block pertaining to 1 normal point into a matrix.
- subroutine setup_local_block (nm_x, r_id, ind, block, block_loc)

  /private Set up local block, possibly translated for fast PB3D.

Variables

- logical, public slepc_utilities::debug_insert_block_mat = .false.

  plot debug information for insert_block_mat()

D.74.1 Function/Subroutine Documentation

D.74.1.1 setup_local_block()

```
subroutine insert_block_mat::setup_local_block (
            dimension(:,:), intent(in) nm_x,
            intent(in) r_id,
            dimension(2), intent(in) ind,
            dimension(:,:), intent(in) block,
            dimension(:,:), intent(inout) block_loc)
```

/private Set up local block, possibly translated for fast PB3D.

Note

For BC_style 3, there is a grid extension of n_r beyond the tabulated modes variables, which is why there is a 'min(..,size(nm_X,1))' here.

Definition at line 207 of file SLEPC_utilities.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

D.75 SLEPC_utilities.f90

Go to the documentation of this file.

```
00001 !-----------------------------------------------------------------------!
00002 !> Numerical utilities related to SLEPC (and PETSC) operations.
00003 !-----------------------------------------------------------------------!
00004 !> \see
00005 !! References:
00006 !! \cite Hernandez2005slepc
00007 !-----------------------------------------------------------------------!
00008 module slepc_utilities
00009 #include <PB3D_macros.h>
00010 #include <wrappers.h>
00011 #include <slepc/finclude/slepceps.h>
00012 ! for slepc 3.6.0:
00013 !#include <slepc/finclude/slepcepsdef.h>
00014 ! for slepc 3.5.3:
00015 !#include <finclude/slepcepsdef.h>
```

```
    use str_utilities
    use output_ops
    use messages
    use slepceps
    use num_vars, only: iu, dp, max_str_ln

    implicit none
    private
    public insert_block_mat
#if ldebug
    public debug_insert_block_mat
#endif

    ! global variables
#if ldebug
    logical :: debug_insert_block_mat = .false.                                !< plot debug information for insert_block_mat() \ldebug
#endif

contains
    !> Insert a block pertaining to 1 normal point into a matrix.
    !!
    !! Optionally, also set  the Hermitian transpose and /  or overwrite instead
    !! of add value.
    !!
    !! This routine  takes into  account that  if the mode  numbers change  as a
    !! function of the  normal coordinate, as is the case  for X_style 2 (fast),
    !! part of  the local  blocks, which  were set up  without taking  this into
    !! account, can  have information that  correspond to mode numbers  that are
    !! not present any  more for off-diagonal entries, and lack  an equal amount
    !! of information that corresponds to the mode numbers that have taken their
    !! places.
    !!
    !! Omission  of this  effect  is justifiable  if there  are  many more  mode
    !! couplings that are  not omitted (i.e. when \c n_mod_X  is large), so that
    !! it is  a small effect,  and/or when the  coupling between these  modes is
    !! already negligible.
    !!
    !! Unfortunately, it can  be argued that this is not  generally the case for
    !! the  fast  version (\c  X_style  =  2) since  \f$\widetilde{PV}^1\f$  and
    !! \f$\widetilde{PV}^2\f$ are  both \f$\sim (nq-m)\f$, which  indicates that
    !! the smallest elements in the corresponding off-diagonal blocks lie in the
    !! central columns for \f$\widetilde{PV}^1\f$ and in the central columns and
    !! rows  for  \f$\widetilde{PV}^2\f$,  while the  problematic  elements  lie
    !! farthest away from the central columns for \f$\widetilde{PV}^1\f$ and the
    !! central  columns  and  rows  for  \f$\widetilde{PV}^2\f$,  so  these  are
    !! generally not small.
    !!
    !! The coefficients due to normal  discretization get increasingly small for
    !! higher orders, though, which compensates this for higher orders.
    !!
    !! Finally, an alternative  would be to set the problematic  elements not to
    !! zero but  extrapolated from  the closest  elements, but  then consistency
    !! would  be sacrified,  for example  in  the energy  reconstruction of  the
    !! POST-processing: In  the energy reconstruction  the exact same  terms are
    !! neglected as well.
    !!
    !! To conclude, for fast \c X_style 2,  the number of modes (\c n_mod_X) has
    !! to be chosen high enough compared with the variation of the mode numbers;
    !! i.e. the variation of the safety factor or rotational transform.
    !!
    !! \note The grid in which the modes  variables \c mds are tabulated up must
    !! be the  same as the one  in which \c  mat is set  up. If this is  not the
    !! case, the indices in \c mds have to be adapted.
    !!
    !! \return ierr
    integer function insert_block_mat(mds,block,mat,r_id,ind,n_r,transp,&
        &overwrite,ind_insert) result(ierr)
        use x_vars, only: modes_type
        use num_vars, only: use_pol_flux_f
#if ldebug
        use num_vars, only: rank
        use input_utilities, only: pause_prog
#endif

        character(*), parameter :: rout_name = 'insert_block_mat'

        ! input / output
        type(modes_type), intent(in), target :: mds                            !< general modes variables
        petscscalar, intent(in) :: block(:,:)                                  !< (\cn_mod x \cn_mod) block matrix for 1 normal point
        mat, intent(inout) :: mat                                              !< matrix in which to insert block
        petscint, intent(in) :: r_id                                           !< normal position of corresponding \f$\widetilde{V}^0\f$ (starting at 0)
        petscint, intent(in) :: ind(2)                                         !< 2D index in matrix, relative to \cr_id
        petscint, intent(in) :: n_r                                            !< number of grid points of solution grid
        petscbool, intent(in), optional :: transp                              !< also set Hermitian transpose
```

```
        petscbool, intent(in), optional :: overwrite                    !< overwrite
        petscbool, intent(in), optional :: ind_insert                   !< individual insert, only important for debugging

        ! local variables
        petscint, pointer :: nm_x(:,:)                                  ! m (pol. flux) or n (tor. flux)
        petscbool :: transp_loc                                         ! local copy of transp
        petscbool :: overwrite_loc                                      ! local copy of overwrite
        character(len=max_str_ln) :: err_msg                            ! error message
        petscint :: operation                                           ! either ADD_VALUES or INSERT_VALUES
        petscscalar, allocatable :: block_loc(:,:)                      ! local block, possibly shifted from block
        petscbool :: ind_insert_loc                                     ! local ind_insert

        ! initialize ierr
        ierr = 0

        ! set up local transp, overwrite and ind_insert
        transp_loc = .false.
        if (present(transp)) transp_loc = transp
        overwrite_loc = .false.
        if (present(overwrite)) overwrite_loc = overwrite
        ind_insert_loc = .false.
        if (present(ind_insert)) ind_insert_loc = ind_insert

        ! set operation
        if (overwrite_loc) then
            operation = insert_values
        else
            operation = add_values
        end if

        ! set nm_X
        if (use_pol_flux_f) then
            nm_x => mds%m
        else
            nm_x => mds%n
        end if

#if ldebug
        ! user output
        if (debug_insert_block_mat) then
            if (.not.ind_insert_loc) call sleep(rank)
            call writo('>>> rank '//trim(i2str(rank))//&
                &': at (k,m) = '//trim(i2str(r_id))//' + ('//&
                &trim(i2str(ind(1)))//','//trim(i2str(ind(2)))//'):',&
                &persistent=.true.)
            call lvl_ud(1)
        end if
#endif

        ! only set values if within matrix range
        if (minval(r_id+ind).ge.0 .and. maxval(r_id+ind).lt.n_r) then
            ! set error message
            err_msg = 'Couldn''t add values to matrix'

            ! initialize
            allocate(block_loc(size(block,1),size(block,2)))

            ! untransposed block
            call setup_local_block(nm_x,r_id,ind(1:2),block,block_loc)

            ! set values
            call matsetvaluesblocked(mat,1,r_id+ind(1),1,r_id+ind(2),&
                &transpose(block_loc),operation,ierr)
            chckerr(err_msg)

            ! untransposed block
            if (transp_loc) then
#if ldebug
                if (debug_insert_block_mat) then
                    call writo('Also at the transposed place:')
                end if
#endif

                ! transposed block
                call setup_local_block(nm_x,r_id,ind(2:1:-1),&
                    &transpose(conjg(block)),block_loc)

                call matsetvaluesblocked(mat,1,r_id+ind(2),1,r_id+ind(1),&
                    &transpose(block_loc),operation,ierr)
                chckerr(err_msg)
            end if

            ! clean up
            deallocate(block_loc)
        else
#if ldebug
```

```
            if (debug_insert_block_mat) &
                &call writo('Due to out of range no local block is added',&
                &persistent=.true.)
#endif
        end if

        ! clean up
        nullify(nm_x)

#if ldebug
        if (debug_insert_block_mat) then
            call pause_prog(ind_insert_loc)
            call lvl_ud(-1)
        endif
#endif
    contains
        !> /private Set up local block, possibly translated for fast PB3D.
        !!
        !! \note  For  BC_style  3,  there  is  a  grid  extension  of  n_r
        !! beyond  the  tabulated modes  variables,  which  is  why there  is  a
        !! 'min(..,size(nm_X,1))' here.
        subroutine setup_local_block(nm_X,r_id,ind,block,block_loc)
            use x_vars, only: n_mod_x

            ! input / output
            petscint, intent(in) :: nm_x(:,:)                                   !< m (pol. flux) or n (tor. flux)
            petscint, intent(in) :: r_id                                        !< normal position of corresponding
    \f$\widetilde{V}^0\f$ (starting at 0)
            petscint, intent(in) :: ind(2)                                      !< 2D index in matrix, relative to \cr_id
            petscscalar, intent(in) :: block(:,:)                               !< n_mod_X x n_mod_X block
            petscscalar, intent(inout) :: block_loc(:,:)                        !< adapted block

            ! local variables
            petscint :: k, m                                                   ! counters
            petscint :: k_loc, m_loc                                            ! local k and m

            block_loc = 0._dp
            do m = 1,n_mod_x
                do k = 1,n_mod_x
                    k_loc = k + nm_x(r_id+1,k) - &
                        &nm_x(min(r_id+1+ind(1),size(nm_x,1)),k)
                    m_loc = m + nm_x(r_id+1,m) - &
                        &nm_x(min(r_id+1+ind(2),size(nm_x,1)),m)
                    if (k_loc.ge.1 .and. m_loc.ge.1 .and. &
                        &k_loc.le.n_mod_x .and. m_loc.le.n_mod_x) &
                        block_loc(k_loc,m_loc) = block(k,m)
                end do
            end do

#if ldebug
            if (debug_insert_block_mat) then
                call writo('following local block is going to be added by &
                    &rank '//trim(i2str(rank))//':',persistent=.true.)
                call writo('Re =',persistent=.true.)
                call print_ar_2(rp(block_loc))
                call writo('Im =',persistent=.true.)
                call print_ar_2(ip(block_loc))
                if (overwrite_loc) then
                    call writo('(with operation INSERT_VALUES)',&
                        &persistent=.true.)
                else
                    call writo('(with operation ADD_VALUES)',&
                        &persistent=.true.)
                end if
            endif
#endif
        end subroutine setup_local_block
    end function insert_block_mat
end module slepc_utilities
```

D.76 /Users/toonweyens/Code/PB3D/Modules/sol_ops.f90 File Reference

Modules

- module sol_ops

  Operations on the solution vectors such as decompososing the energy, etc...

Functions/Subroutines

- subroutine, public sol_ops::plot_sol_vals (sol, last_unstable_id)

  Plots Eigenvalues.
- integer function, public sol_ops::plot_sol_vec (mds_x, mds_sol, grid_eq, grid_x, grid_sol, eq_1, eq_2, x, sol, xyz, x_id, plot_var)

  Plots Eigenvectors.
- integer function, public sol_ops::plot_harmonics (mds, grid_sol, sol, x_id, res_surf)

  Plots the harmonics and their maximum in 2-D.
- integer function, public sol_ops::decompose_energy (mds_x, mds_sol, grid_eq, grid_x, grid_sol, eq_1, eq_2, x, sol, vac, x_id, b_aligned, xyz, e_pot_int, e_kin_int)

  Decomposes the plasma potential and kinetic energy in its individual terms for an individual Eigenvalue.
- integer function, public sol_ops::print_output_sol (grid, sol, data_name, rich_lvl, remove_previous_arrs)

  Print solution quantities to an output file:

Variables

- logical, public sol_ops::debug_plot_sol_vec = .false.

  plot debug information for plot_sol_vec()
- logical, public sol_ops::debug_calc_e = .false.

  plot debug information for calc_E()
- logical, public sol_ops::debug_x_norm = .false.

  plot debug information X_norm
- logical, public sol_ops::debug_du = .false.

  plot debug information for calculation of DU

D.77 sol_ops.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Operations on the solution vectors such as decompososing the energy, etc...
!-----------------------------------------------------------------------------!
module sol_ops
#include <PB3D_macros.h>
#include <wrappers.h>
    use str_utilities
    use output_ops
    use messages
    use num_vars, only: dp, iu, max_str_ln, pi, rank
    use grid_vars, only: grid_type
    use eq_vars, only: eq_1_type, eq_2_type
    use x_vars, only: x_1_type, modes_type
    use sol_vars, only: sol_type
    use vac_vars, only: vac_type

    implicit none
    private
    public plot_harmonics, plot_sol_vec, decompose_energy, print_output_sol, &
        &plot_sol_vals
#if ldebug
    public debug_plot_sol_vec, debug_calc_e, debug_du, debug_x_norm
#endif

    ! global variables
#if ldebug
    logical :: debug_plot_sol_vec = .false.                                 !< plot debug information for plot_sol_vec() \ldebug
    logical :: debug_plot_harmonics = .false.                               !< plot debug information for plot_harmonics() \ldebug
```

```
    logical :: debug_calc_e = .false.                                   !< plot debug information for calc_E() \ldebug
    logical :: debug_x_norm = .false.                                   !< plot debug information \c X_norm \ldebug
    logical :: debug_du = .false.                                       !< plot debug information for calculation of \c DU \ldebug
#endif

contains
    !> Plots Eigenvalues.
    subroutine plot_sol_vals(sol,last_unstable_id)
        use num_vars, only: rank

        ! input / output
        type(sol_type), intent(in) :: sol                               !< solution variables
        integer, intent(in) :: last_unstable_id                         !< index of last unstable EV

        ! local variables
        character(len=max_str_ln) :: plot_title                         ! title for plots
        character(len=max_str_ln) :: plot_name                          ! file name for plots
        integer :: n_sol_found                                          ! how many solutions found and saved
        integer :: id                                                   ! counter

        ! set local variables
        n_sol_found = size(sol%val)

        ! only let master plot
        if (rank.eq.0) then
            ! Last Eigenvalues
            plot_title = 'final Eigenvalues omega^2 [log]'
            plot_name = 'Eigenvalues'
            call print_ex_2d(plot_title,plot_name,&
                &log10(abs(rp(sol%val(1:n_sol_found)))),draw=.false.)
            call draw_ex([plot_title],plot_name,1,1,.false.,ex_plot_style=1)

            ! Last Eigenvalues: unstable range
            if (last_unstable_id.gt.0) then
                plot_title = 'final unstable Eigenvalues omega^2'
                plot_name = 'Eigenvalues_unstable'
                call print_ex_2d(plot_title,plot_name,&
                    &rp(sol%val(1:last_unstable_id)),&
                    &x=[(id*1._dp,id=1,last_unstable_id)],draw=.false.)
                call draw_ex([plot_title],plot_name,1,1,.false.,ex_plot_style=1)
            end if

            ! Last Eigenvalues: stable range
            if (last_unstable_id.lt.n_sol_found) then
                plot_title = 'final stable Eigenvalues omega^2'
                plot_name = 'Eigenvalues_stable'
                call print_ex_2d(plot_title,plot_name,&
                    &rp(sol%val(last_unstable_id+1:n_sol_found)),&
                    &x=[(id*1._dp,id=last_unstable_id+1,n_sol_found)],&
                    &draw=.false.)
                call draw_ex([plot_title],plot_name,1,1,.false.,ex_plot_style=1)
            end if
        end if
    end subroutine plot_sol_vals

    !> Plots Eigenvectors.
    !!
    !! This is done using the angular  part of the the provided equilibrium grid
    !! and the normal part of the provided solution grid.
    !!
    !! The perturbation grid is assumed to  have the same angular coordinates as
    !! the equilibrium grid, and the normal coordinates correspond to either the
    !! equilibrium grid (\c X_grid_style 1),  the solution grid (\c X_grid_style
    !! 2) or the enriched equilibrium grid (\c X_grid_style 3).
    !!
    !! The output grid,  furthermore, has the angular part  corresponding to the
    !! equilibrium grid, and the normal part to the solution grid.
    !!
    !! \note The normalization factors are taken  into account and the output is
    !! transformed back to unnormalized values:
    !!  \f[\begin{aligned}
    !!      \vec{\xi} &\sim \frac{X_0}{R_0 B_0} \ , \\
    !!      \vec{Q}   &\sim \frac{X_0}{R_0^2} \ , \\
    !!  \end{aligned}\f]
    !!
    !! which translates to
    !!  \f[\begin{aligned}
    !!      X  &\sim X_0 \ , \\
    !!      U  &\sim \frac{X_0}{R_0^2 B_0} \ , \\
    !!      Qn &\sim \frac{X_0 B_0}{R_0} \ , \\
    !!      Qg &\sim \frac{X_0}{R_0^3} \ ,
    !!  \end{aligned}\f]
    !! where \f$X_0\f$ is not determined but is common to all factors.
    !!
    !! \return ierr
```

```
    integer function plot_sol_vec(mds_X,mds_sol,grid_eq,grid_X,grid_sol,eq_1,&
        &eq_2,X,sol,XYZ,X_id,plot_var) result(ierr)

        use num_vars, only: no_plots, tol_zero, pert_mult_factor_post, &
            &eq_job_nr, eq_jobs_lims, eq_job_nr, norm_disc_prec_eq, &
            &norm_disc_prec_x, norm_disc_prec_sol, use_normalization, &
            &x_grid_style, eq_style
        use grid_utilities, only: trim_grid, calc_vec_comp
        use sol_utilities, only: calc_xuq
        use eq_vars, only: r_0, b_0
        use num_utilities, only: c, spline
        use mpi_utilities, only: get_ser_var
#if ldebug
        use num_vars, only: use_pol_flux_f
#endif

        character(*), parameter :: rout_name = 'plot_sol_vec'

        ! input / output
        type(modes_type), intent(in) :: mds_x                                   !< general modes variables in perturbation grid
        type(modes_type), intent(in) :: mds_sol                                 !< general modes variables in solution grid
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(grid_type), intent(in) :: grid_x                                   !< perturbation grid
        type(grid_type), intent(in) :: grid_sol                                 !< solution grid
        type(eq_1_type), intent(in) :: eq_1                                     !< flux equilibrium
        type(eq_2_type), intent(in) :: eq_2                                     !< metric equilibrium
        type(x_1_type), intent(in) :: x                                         !< perturbation variables
        type(sol_type), intent(in) :: sol                                       !< solution variables
        real(dp), intent(in) :: xyz(:,:,:,:)                                    !< X, Y and Z of extended eq_grid
        integer, intent(in) :: x_id                                             !< nr. of Eigenvalue (for output name)
        logical, intent(in) :: plot_var(2)                                      !< whether variables are plotted (1: plasma perturbation, 2: magnetic perturbation)

        ! local variables
        type(grid_type) :: grid_out                                             ! output grid (see description)
        type(grid_type) :: grid_out_trim                                        ! trimmed output grid
        integer :: norm_id(2)                                                   ! untrimmed normal indices for trimmed grids
        integer :: id, jd, jd2, kd, td                                          ! counters
        integer :: n_t(2)                                                       ! nr. of time steps in quarter period, nr. of quarter periods
        integer :: plot_dim(4)                                                  ! dimensions of plot
        integer :: plot_offset(4)                                               ! local offset of plot
        integer :: col                                                          ! collection type for HDF5 plot
        logical :: cont_plot                                                    ! continued plot
        logical :: plot_var_loc(2)                                              ! local plot_var
        logical :: xyz_plot_setup                                               ! XYZ_plot_setup has been set up
        real(dp) :: x_0                                                         ! X at theta = zeta = 0
        real(dp), allocatable :: x_0_ser(:)                                     ! X_0 for all processes
        real(dp), allocatable :: time(:)                                        ! fraction of Alfvén time
        real(dp), allocatable :: xyz_plot(:,:,:,:,:)                            ! copies of XYZ
        real(dp), allocatable :: xyz_vec(:,:,:,:,:)                             ! copies of XYZ for vector plot
        real(dp), allocatable :: h12(:,:,:)                                     ! interpolated h_FD(1,2) on solution grid
        real(dp), allocatable :: h22(:,:,:)                                     ! interpolated h_FD(2,2) on solution grid
        real(dp), allocatable :: h23(:,:,:)                                     ! interpolated h_FD(3,2) on solution grid
        real(dp), allocatable :: g13(:,:,:)                                     ! interpolated g_FD(1,3) on solution grid
        real(dp), allocatable :: g33(:,:,:)                                     ! interpolated g_FD(3,3) on solution grid
        real(dp), allocatable :: jac_fd_int(:,:,:)                              ! interpolated jac_FD on solution grid
        real(dp), allocatable :: f_plot_phase(:,:,:,:)                          ! phase of f_plot
        real(dp), allocatable :: ccomp(:,:,:,:,:)                               ! Cart. components of perturbation
        complex(dp) :: omega                                                    ! sqrt of Eigenvalue
        complex(dp), allocatable :: f_plot(:,:,:,:,:)                           ! the function to plot
        character(len=max_str_ln) :: err_msg                                    ! error message
        character(len=max_str_ln) :: var_name(2)                                ! name of variable that is plot
        character(len=max_str_ln) :: file_name(2)                               ! name of file
        character(len=max_str_ln) :: description(2)                             ! description
        character(len=2) :: sol_name                                            ! name of solution vector ('xi' or 'Q')
#if ldebug
        integer :: nm                                                           ! n (pol. flux) or m (tor. flux)
        integer :: ld                                                           ! counter
        real(dp), allocatable :: dq_saf(:)                                      ! norm. deriv. of q_saf interpolated on solution grid
        real(dp), allocatable :: drot_t(:)                                      ! norm. deriv. of rot_t interpolated on solution grid
        complex(dp), allocatable :: u_inf(:,:,:,:)                              ! ideal ballooning U
        complex(dp), allocatable :: u_inf_prop(:,:,:)                           ! proportional part of U_inf
#endif

        ! initialize ierr
        ierr = 0

        ! bypass plots if no_plots
        if (no_plots) return

        ! return if no variables are plot
        if (count(plot_var).eq.0) return
        plot_var_loc = plot_var
```

```
        if (abs(pert_mult_factor_post).ge.tol_zero) then
            ! set up X_0
            x_0 = 0._dp
            do kd = 1,grid_sol%loc_n_r
                x_0 = max(x_0,abs(sum(sol%vec(:,kd,x_id))))
            end do
            ierr = get_ser_var([x_0],x_0_ser,scatter=.true.)
            chckerr('')
            x_0 = maxval(x_0_ser)

            ! user output
            call writo('Perturbing the position vector (X,Y,Z) with &
                &multiplicative factor '//trim(r2strt(pert_mult_factor_post)))
            if (.not.plot_var(1)) then
                call writo('For nonzero "pert_mult_factor_POST", need to also &
                    &plot xi',warning=.true.)
                plot_var_loc(1) = .true.
            end if
        end if

        ! user output
        call writo('Plot the solution vector')
        call lvl_ud(1)

#if ldebug
        ! tests
        if (size(xyz,4).ne.3) then
            ierr = 1
            err_msg = 'X, Y and Z needed'
            chckerr(err_msg)
        end if
        if (grid_eq%n(1).ne.size(xyz,1) .or. &
            &grid_eq%n(2).ne.size(xyz,2) .or. &
            &grid_sol%loc_n_r.ne.size(xyz,3)) then
            ierr = 1
            err_msg = 'XYZ needs to have the correct dimensions'
            chckerr(err_msg)
        end if
#endif

        ! set up n_t
        ! if the  Eigenvalue is negative,  the Eigenfunction explodes,  so limit
        ! n_t(2) to 1. If it is positive, the Eigenfunction oscilates, so choose
        ! n_t(2) = 8 for 2 whole periods
        if (rp(sol%val(x_id)).lt.0) then                                    ! exploding, unstable
            n_t(1) = 1                                                      ! 1 point per quarter period
            n_t(2) = 1                                                      ! 1 quarter period
        else                                                                ! oscillating, stable
            n_t(1) = 2                                                      ! 2 points per quarter period
            n_t(2) = 4                                                      ! 4 quarter periods
        end if

        ! set collection type
        if (product(n_t).gt.1) then
            col = 2                                                         ! temporal collection
        else
            col = 1                                                         ! no collection
        end if

        ! set up output grid
        ierr = grid_out%init([grid_eq%n(1:2),grid_sol%n(3)],&
            &i_lim=[grid_sol%i_min,grid_sol%i_max],divided=grid_sol%divided)
        chckerr('')
        grid_out%r_F = grid_sol%r_F
        grid_out%r_E = grid_sol%r_E
        grid_out%loc_r_F = grid_sol%loc_r_F
        grid_out%loc_r_E = grid_sol%loc_r_E
        if (eq_style.eq.1) allocate(grid_out%trigon_factors(&
            &size(grid_x%trigon_factors,1),size(grid_x%trigon_factors,2),&
            &size(grid_x%trigon_factors,3),grid_out%loc_n_r,&
            &size(grid_x%trigon_factors,5)))                                ! VMEC
        select case (x_grid_style)
            case (1,3)                                                      ! equilibrium or enriched
                ! interpolate X->sol
                do jd = 1,grid_x%n(2)
                    do id = 1,grid_x%n(1)
                        ierr = spline(grid_x%loc_r_F,grid_x%theta_F(id,jd,:),&
                            &grid_sol%loc_r_F,grid_out%theta_F(id,jd,:),&
                            &ord=norm_disc_prec_x)
                        chckerr('')
                        ierr = spline(grid_x%loc_r_F,grid_x%theta_E(id,jd,:),&
                            &grid_sol%loc_r_F,grid_out%theta_E(id,jd,:),&
                            &ord=norm_disc_prec_x)
                        chckerr('')
                        ierr = spline(grid_x%loc_r_F,grid_x%zeta_F(id,jd,:),&
                            &grid_sol%loc_r_F,grid_out%zeta_F(id,jd,:),&
                            &ord=norm_disc_prec_x)
```

```
                        chckerr('')
                        ierr = spline(grid_x%loc_r_F,grid_x%zeta_E(id,jd,:),&
                            &grid_sol%loc_r_F,grid_out%zeta_E(id,jd,:),&
                            &ord=norm_disc_prec_x)
                        chckerr('')
                        if (eq_style.eq.1) then                                 ! VMEC
                            do td = 1,size(grid_x%trigon_factors,5)
                                do kd = 1,size(grid_x%trigon_factors,1)
                                    ierr = spline(grid_x%loc_r_F,&
                                        &grid_x%trigon_factors(kd,id,jd,:,td),&
                                        &grid_sol%loc_r_F,&
                                        &grid_out%trigon_factors&
                                        &(kd,id,jd,:,td),ord=norm_disc_prec_x)
                                    chckerr('')
                                end do
                            end do
                        end if
                    end do
                end do
            case (2)                                                            ! solution
                ! copy
                grid_out%theta_F = grid_x%theta_F
                grid_out%theta_E = grid_x%theta_E
                grid_out%zeta_F = grid_x%zeta_F
                grid_out%zeta_E = grid_x%zeta_E
        end select

        ! trim output grid
        ierr = trim_grid(grid_out,grid_out_trim,norm_id)
        chckerr('')

        ! set up plot dimensions and offset
        plot_dim = [grid_out_trim%n,product(n_t)]
        plot_offset = [0,0,grid_out_trim%i_min-1,0]

        ! possibly modify if multiple equilibrium parallel jobs
        if (size(eq_jobs_lims,2).gt.1) then
            plot_dim(1) = eq_jobs_lims(2,size(eq_jobs_lims,2)) - &
                &eq_jobs_lims(1,1) + 1
            plot_offset(1) = eq_jobs_lims(1,eq_job_nr) - 1
        end if

        ! set up continued plot
        cont_plot = eq_job_nr.gt.1

        ! For each time step, calculate the time (as fraction of Alfvén time)
        allocate(time(product(n_t)))
        do td = 1,product(n_t)
            if (n_t(1).eq.1) then
                time(td) = (td-1) * 0.25
            else
                time(td) = (mod(td-1,n_t(1))*1._dp/n_t(1) + &
                    &(td-1)/n_t(1)) * 0.25
            end if
        end do

        ! set up interpolated metric factors (not trimmed)
        allocate(h12(grid_out%n(1),grid_out%n(2),grid_out%loc_n_r))
        allocate(h22(grid_out%n(1),grid_out%n(2),grid_out%loc_n_r))
        allocate(h23(grid_out%n(1),grid_out%n(2),grid_out%loc_n_r))
        allocate(g13(grid_out%n(1),grid_out%n(2),grid_out%loc_n_r))
        allocate(g33(grid_out%n(1),grid_out%n(2),grid_out%loc_n_r))
        allocate(jac_fd_int(grid_out%n(1),grid_out%n(2),grid_out%loc_n_r))
#if ldebug
        allocate(dq_saf(grid_out%loc_n_r))
        allocate(drot_t(grid_out%loc_n_r))
#endif

        ! interpolate eq->sol
        do jd = 1,grid_eq%n(2)
            do id = 1,grid_eq%n(1)
                ierr = spline(grid_eq%loc_r_F,&
                    &eq_2%h_FD(id,jd,:,c([1,2],.true.),0,0,0),grid_sol%loc_r_F,&
                    &h12(id,jd,:),ord=norm_disc_prec_eq)
                chckerr('')
                ierr = spline(grid_eq%loc_r_F,&
                    &eq_2%h_FD(id,jd,:,c([2,2],.true.),0,0,0),grid_sol%loc_r_F,&
                    &h22(id,jd,:),ord=norm_disc_prec_eq)
                chckerr('')
                ierr = spline(grid_eq%loc_r_F,&
                    &eq_2%h_FD(id,jd,:,c([2,3],.true.),0,0,0),grid_sol%loc_r_F,&
                    &h23(id,jd,:),ord=norm_disc_prec_eq)
                chckerr('')
                ierr = spline(grid_eq%loc_r_F,&
                    &eq_2%g_FD(id,jd,:,c([1,3],.true.),0,0,0),grid_sol%loc_r_F,&
                    &g13(id,jd,:),ord=norm_disc_prec_eq)
                chckerr('')
```

```
                ierr = spline(grid_eq%loc_r_F,&
                    &eq_2%g_FD(id,jd,:,c([3,3],.true.),0,0,0),grid_sol%loc_r_F,&
                    &g33(id,jd,:),ord=norm_disc_prec_eq)
                chckerr('')
                ierr = spline(grid_eq%loc_r_F,&
                    &eq_2%jac_FD(id,jd,:,0,0,0),grid_sol%loc_r_F,&
                    &jac_fd_int(id,jd,:),ord=norm_disc_prec_eq)
                chckerr('')
            end do
        end do
#if ldebug
        ierr = spline(grid_eq%loc_r_F,eq_1%q_saf_FD(:,1),grid_sol%loc_r_F,&
            &dq_saf,ord=norm_disc_prec_eq)
        chckerr('')
        ierr = spline(grid_eq%loc_r_F,eq_1%rot_t_FD(:,1),grid_sol%loc_r_F,&
            &drot_t,ord=norm_disc_prec_eq)
        chckerr('')
#endif

        ! calculate omega =  sqrt(sol_val) and make sure to  select the decaying
        ! solution
        omega = sqrt(sol%val(x_id))
        if (ip(omega).gt.0) omega = - omega                                  ! exploding solution, not the decaying one

        ! set up f_plot (not trimmed) and XYZ_plot (trimmed)
        ! set up:
        !   * f_plot (not trimmed): X, U, Q
        !   * ccomp (not trimmed): Cartesian components of perturbation
        !   * XYZ_plot (trimmed): copies of XYZ for plot for different times
        !   * XYZ_vec (trimmed): copies of XYZ for vector plot
        allocate(f_plot(grid_out%n(1),grid_out%n(2),grid_out%loc_n_r,&
            &product(n_t),2))
        allocate(ccomp(grid_out%n(1),grid_out%n(2),grid_out%loc_n_r,3,2))
        allocate(xyz_plot(grid_out%n(1),grid_out%n(2),grid_out_trim%loc_n_r,&
            &size(time),3))
        allocate(xyz_vec(grid_out%n(1),grid_out%n(2),grid_out_trim%loc_n_r,3,&
            &3))
        xyz_plot_setup = .false.

        ! operations for plasma perturbation and magnetic perturbation
        do jd = 1,2                                                          ! first plasma perturbation, then magnetic perturbation
            if (.not.plot_var_loc(jd)) cycle

            ! calculate vector components and  set solution name, variable name,
            ! file name and description
            select case (jd)
                case (1)                                                     ! plasma perturbation
                    ierr = calc_xuq(mds_x,mds_sol,grid_eq,grid_x,grid_sol,eq_1,&
                        &eq_2,x,sol,x_id,1,time,f_plot(:,:,:,:,1))
                    chckerr('')
                    ierr = calc_xuq(mds_x,mds_sol,grid_eq,grid_x,grid_sol,eq_1,&
                        &eq_2,x,sol,x_id,2,time,f_plot(:,:,:,:,2))
                    chckerr('')
                    sol_name = 'xi'
                    file_name(1) = trim(i2str(x_id))//'_sol_X'
                    file_name(2) = trim(i2str(x_id))//'_sol_U'
                case (2)                                                     ! magnetic perturbation
                    ierr = calc_xuq(mds_x,mds_sol,grid_eq,grid_x,grid_sol,eq_1,&
                        &eq_2,x,sol,x_id,3,time,f_plot(:,:,:,:,1))
                    chckerr('')
                    ierr = calc_xuq(mds_x,mds_sol,grid_eq,grid_x,grid_sol,eq_1,&
                        &eq_2,x,sol,x_id,4,time,f_plot(:,:,:,:,2))
                    chckerr('')
                    sol_name = 'Q'
                    file_name(1) = trim(i2str(x_id))//'_sol_Qn'
                    file_name(2) = trim(i2str(x_id))//'_sol_Qg'
            end select
            var_name(1) = 'Normal comp. of '//trim(sol_name)
            var_name(2) = 'Geodesic comp. of '//trim(sol_name)
            description(1) = 'Normal component of vector '//trim(sol_name)//&
                &' for Eigenvalue '//trim(i2str(x_id))//' with omega = '//&
                &trim(r2str(rp(omega)))
            description(2) = 'Geodesic component of vector '//trim(sol_name)//&
                &' for Eigenvalue '//trim(i2str(x_id))//' with omega = '//&
                &trim(r2str(rp(omega)))

            ! calculate vector components at every time point
            do td = 1,product(n_t)
                ! covariant components:
                !   xi . e_alpha = U J^2 h22/g33
                !   xi . e_psi   = X 1/h22 - U J^2 h12/g33
                !   xi . e_theta = 0
                ! and similar for Q
                ccomp(:,:,:,1,1) = rp(f_plot(:,:,:,td,2)) * &
                    &jac_fd_int**2*h22/g33
                ccomp(:,:,:,2,1) = rp(f_plot(:,:,:,td,1)) * &
```

```
                    &1._dp/h22 - &
                    &rp(f_plot(:,:,:,td,2)) * &
                    &jac_fd_int**2*h12/g33
                ccomp(:,:,:,3,1) = 0._dp

                ! contravariant components:
                !   xi . nabla alpha = X h12/h22 + U
                !   xi . nabla psi   = X
                !   xi . nabla theta = X h23/h22 - U g13/g33
                ! and similar for Q
                ccomp(:,:,:,1,2) = rp(f_plot(:,:,:,td,1)) * &
                    &h12/h22 + rp(f_plot(:,:,:,td,2))
                ccomp(:,:,:,2,2) = rp(f_plot(:,:,:,td,1))
                ccomp(:,:,:,3,2) = rp(f_plot(:,:,:,td,1)) * &
                    &h23/h22 - rp(f_plot(:,:,:,td,2)) * g13/g33

                ! multiplication factor if xi
                if (abs(pert_mult_factor_post).ge.tol_zero .and. jd.eq.1) &
                    &ccomp = ccomp*pert_mult_factor_post/x_0

                ! transform normalized quantities to unnormalized
                if (use_normalization) then
                    select case (jd)
                        case (1)                                        ! plasma perturbation
                            ccomp = ccomp / (r_0*b_0)                   ! xi ~ X_0/(R_0*B_0)
                        case (2)                                        ! magnetic perturbation
                            ccomp = ccomp / (r_0**2)                    ! Q ~ X_0/(R_0^2)
                    end select
                end if

                ! transform to Cartesian components
                ierr = calc_vec_comp(grid_out,grid_eq,eq_1,eq_2,ccomp,&
                    &norm_disc_prec_sol)
                chckerr('')

                ! vector plot
                do id = 1,3
                    xyz_vec(:,:,:,id,:) = xyz(:,:,norm_id(1):norm_id(2),:)
                end do
                call plot_hdf5([trim(sol_name)],trim(sol_name)//'_T_'//&
                    &trim(i2str(td)),ccomp(:,:,norm_id(1):norm_id(2),:,1),&
                    &tot_dim=[plot_dim(1:3),3],&
                    &loc_offset=[plot_offset(1:3),0],&
                    &x=xyz_vec(:,:,:,:,1),y=xyz_vec(:,:,:,:,2),&
                    &z=xyz_vec(:,:,:,:,3),col=4,cont_plot=cont_plot,&
                    &descr='perturbation for time t = '//&
                    &trim(r2strt(time(td))))

                ! perturb position vector if xi;  it does not matter whether
                ! we take covariant or contravariant components
                if (.not.xyz_plot_setup) then
                    xyz_plot(:,:,:,td,:) = xyz(:,:,norm_id(1):norm_id(2),:)
                    if (abs(pert_mult_factor_post).ge.tol_zero .and. jd.eq.1) &
                        &xyz_plot(:,:,:,td,:) = xyz_plot(:,:,:,td,:) + &
                        &ccomp(:,:,norm_id(1):norm_id(2),:,1)
                end if
            end do

            ! XYZ_plot has been set up if we get here
            xyz_plot_setup = .true.

#if ldebug
            if (debug_plot_sol_vec .and. jd.eq.1) then
                call writo('Checking how well U approximates the pure &
                    &ballooning result')
                call lvl_ud(1)

                ! set up ideal U
                allocate(u_inf(grid_out%n(1),grid_out%n(2),grid_out%loc_n_r,&
                    &product(n_t)))

                ! derive the X vector
                do ld = 1,product(n_t)
                    do jd2 = 1,grid_out%n(2)
                        do id = 1,grid_out%n(1)
                            ierr = spline(grid_out%loc_r_F,&
                                &f_plot(id,jd2,:,ld,1),grid_out%loc_r_F,&
                                &u_inf(id,jd2,:,ld),ord=norm_disc_prec_eq,&
                                &deriv=1)
                            chckerr('')
                        end do
                    end do
                end do

                ! set up dummy variable Theta^alpha + q' theta and nm
                allocate(u_inf_prop(grid_out%n(1),grid_out%n(2),&
                    grid_out%loc_n_r))
```

```
                if (use_pol_flux_f) then
                    do kd = 1,grid_out%loc_n_r
                        u_inf_prop(:,:,kd) = h12(:,:,kd)/h22(:,:,kd) + &
                            &dq_saf(kd)*grid_out%theta_F(:,:,kd)
                    end do
                    nm = sol%n(1,1)
                else
                    do kd = 1,grid_out%loc_n_r
                        u_inf_prop(:,:,kd) = h12(:,:,kd)/h22(:,:,kd) - &
                            &drot_t(kd)*grid_out%zeta_F(:,:,kd)
                    end do
                    nm = sol%m(1,1)
                end if

                ! multiply by i/n or i/m and add the proportional part
                do ld = 1,product(n_t)
                    u_inf(:,:,:,ld) = iu/nm * u_inf(:,:,:,ld) - &
                        &f_plot(:,:,:,ld,1) * u_inf_prop
                end do
                deallocate(u_inf_prop)

                ! plotting real part
                call plot_hdf5('RE X','TEST_RE_X_'//trim(i2str(x_id)),&
                    &rp(f_plot(:,:,norm_id(1):norm_id(2),1,1)),&
                    &tot_dim=plot_dim(1:3),loc_offset=plot_offset(1:3))
                call plot_diff_hdf5(rp(f_plot(:,:,norm_id(1):norm_id(2),1,2)),&
                    &rp(u_inf(:,:,norm_id(1):norm_id(2),1)),&
                    &'TEST_RE_U_inf_'//trim(i2str(x_id)),tot_dim=plot_dim(1:3),&
                    &loc_offset=plot_offset(1:3),descr='To test whether U &
                    &approximates the ideal ballooning result',&
                    &output_message=.true.)

                ! plotting imaginary part
                call plot_hdf5('IM X','TEST_IM_X_'//trim(i2str(x_id)),&
                    &ip(f_plot(:,:,norm_id(1):norm_id(2),1,1)),&
                    &tot_dim=plot_dim(1:3),loc_offset=plot_offset(1:3))
                call plot_diff_hdf5(ip(f_plot(:,:,norm_id(1):norm_id(2),1,2)),&
                    &ip(u_inf(:,:,norm_id(1):norm_id(2),1)),&
                    &'TEST_IM_U_inf_'//trim(i2str(x_id)),tot_dim=plot_dim(1:3),&
                    &loc_offset=plot_offset(1:3),descr='To test whether U &
                    &approximates the ideal ballooning result',&
                    &output_message=.true.)

                call writo('Checking done')
                call lvl_ud(-1)
            end if
#endif

            ! set up temporary variable for phase
            allocate(f_plot_phase(grid_out%n(1),grid_out%n(2),&
                &grid_out_trim%loc_n_r,product(n_t)))

            ! transform normalized quantities to unnormalized
            if (use_normalization) then
                select case (jd)
                    case (1)                                                ! plasma perturbation
                        !f_plot(:,:,:,:,1) = f_plot(:,:,:,:,1)               ! X
                        f_plot(:,:,:,:,2) = f_plot(:,:,:,:,2) / (r_0**2*b_0)   ! U
                    case (2)                                                ! magnetic perturbation
                        f_plot(:,:,:,:,1) = f_plot(:,:,:,:,1) * b_0/r_0        ! Qn
                        f_plot(:,:,:,:,2) = f_plot(:,:,:,:,2) / (r_0**3)       ! Qg
                end select
            end if

            do kd = 1,2
                call plot_hdf5([var_name(kd)],trim(file_name(kd))//'_RE',&
                    &rp(f_plot(:,:,norm_id(1):norm_id(2),:,kd)),&
                    &tot_dim=plot_dim,loc_offset=plot_offset,&
                    &x=xyz_plot(:,:,:,:,1),y=xyz_plot(:,:,:,:,2),&
                    &z=xyz_plot(:,:,:,:,3),col=col,cont_plot=cont_plot,&
                    &descr=description(kd))
                call plot_hdf5([var_name(kd)],trim(file_name(kd))//'_IM',&
                    &ip(f_plot(:,:,norm_id(1):norm_id(2),:,kd)),&
                    &tot_dim=plot_dim,loc_offset=plot_offset,&
                    &x=xyz_plot(:,:,:,:,1),y=xyz_plot(:,:,:,:,2),&
                    &z=xyz_plot(:,:,:,:,3),col=col,cont_plot=cont_plot,&
                    &descr=description(kd))
                f_plot_phase = atan2(&
                    &ip(f_plot(:,:,norm_id(1):norm_id(2),:,kd)),&
                    &rp(f_plot(:,:,norm_id(1):norm_id(2),:,kd)))
                where (f_plot_phase.lt.0) f_plot_phase = f_plot_phase + 2*pi
                call plot_hdf5([var_name(kd)],trim(file_name(kd))//'_PH',&
                    &f_plot_phase,tot_dim=plot_dim,loc_offset=plot_offset,&
                    &x=xyz_plot(:,:,:,:,1),y=xyz_plot(:,:,:,:,2),&
                    &z=xyz_plot(:,:,:,:,3),col=col,cont_plot=cont_plot,&
                    &descr=description(kd))
            end do
```

```
            ! clean up
            deallocate(f_plot_phase)
        end do

        ! clean up
        call grid_out%dealloc()
        call grid_out_trim%dealloc()

        call lvl_ud(-1)
    end function plot_sol_vec

    !> Plots the harmonics and their maximum in 2-D.
    !!
    !! \return ierr
    integer function plot_harmonics(mds,grid_sol,sol,X_id,res_surf) result(ierr)
        use mpi_utilities, only: get_ser_var
        use num_vars, only: ex_max_size, rank, no_plots, use_pol_flux_f
        use eq_vars, only: max_flux_f
        use sol_utilities, only: calc_tot_sol_vec
        use grid_utilities, only: trim_grid
#if ldebug
        use num_vars, only: norm_disc_prec_sol
#endif

        character(*), parameter :: rout_name = 'plot_harmonics'

        ! input / output
        type(modes_type), intent(in) :: mds                                     !< general modes variables in solution grid
        type(grid_type), intent(in) :: grid_sol                                 !< solution grid
        type(sol_type), intent(in) :: sol                                       !< solution variables
        integer, intent(in) :: x_id                                             !< nr. of Eigenvalue (for output name)
        real(dp), intent(in) :: res_surf(:,:)                                   !< resonant surfaces

        ! local variables
        type(grid_type) :: grid_sol_trim                                        ! trimmed sol grid
        integer :: norm_id(2)                                                   ! untrimmed normal indices for trimmed grids
        integer :: id, ld                                                       ! counters
        integer :: ld_loc                                                       ! local ld
        integer :: max_deriv                                                    ! maximum derivative for sol_vec_ser_tot
        integer :: n_mod_tot                                                    ! total number of modes that can resonate
        integer :: n_mod_loc                                                    ! local number of modes that are used in the calculations
        integer :: min_nm_x                                                     ! minimum n or m in total modes
        character(len=max_str_ln) :: file_name                                  ! name of file of plots of this proc.
        character(len=max_str_ln), allocatable :: plot_title(:)                 ! title for plots
        real(dp) :: norm_factor                                                 ! conversion factor max_flux/2pi from flux to normal coordinate
        real(dp), allocatable :: x_plot(:,:)                                    ! x values of plot
        real(dp), allocatable :: y_plot(:,:)                                    ! y values of plot
        complex(dp), allocatable :: sol_vec_ser(:,:)                            ! serial MPI Eigenvector for local number of modes
        complex(dp), allocatable :: sol_vec_dum(:,:)                            ! dummy vector for sol_vec calculations
        complex(dp), allocatable :: sol_vec_ser_tot(:,:,:)                      ! serial MPI Eigenvector for total number of modes and derivatives
        complex(dp), allocatable :: sol_vec_ser_loc(:)                          ! local sol_vec_ser
        real(dp), allocatable :: sol_vec_phase(:,:)                             ! phase of sol_vec
        character :: nm_x                                                       ! n or m
        character(len=max_str_ln) :: err_msg                                    ! error message
#if ldebug
        real(dp), allocatable :: sol_vec_sum(:,:)                               ! for test output
#endif

        ! initialize ierr
        ierr = 0

        ! bypass plots if no_plots
        if (no_plots) return

        ! trim sol grid
        ierr = trim_grid(grid_sol,grid_sol_trim,norm_id)
        chckerr('')

        ! set up local and total nr.  of modes, which can be different for X
        ! style 2 (see discussion in sol_utilities)
        n_mod_loc = sol%n_mod
        n_mod_tot = size(mds%sec,1)
        min_nm_x = minval(mds%sec(:,1),1)

        ! set up serial sol_vec on master
        if (rank.eq.0) &
            &allocate(sol_vec_ser(1:n_mod_loc,1:grid_sol_trim%n(3)))
        do ld = 1,n_mod_loc
            ierr = get_ser_var(sol%vec(ld,norm_id(1):norm_id(2),x_id),&
                &sol_vec_ser_loc,scatter=.true.)
            chckerr('')
            if (rank.eq.0) sol_vec_ser(ld,:) = sol_vec_ser_loc
```

```
            deallocate(sol_vec_ser_loc)
        end do

        ! convert to full mode on master and set up normal factor
        if (rank.eq.0) then
#if ldebug
            select case (norm_disc_prec_sol)
                case (1:2)
                    max_deriv = 1
                case (3)
                    max_deriv = 2
            end select
#else
            max_deriv = 0
#endif
            allocate(sol_vec_ser_tot(1:n_mod_tot,1:grid_sol_trim%n(3),&
                &0:max_deriv))
            do id = 0,max_deriv
                ierr = calc_tot_sol_vec(mds,grid_sol,sol_vec_ser,&
                    &sol_vec_dum,deriv=id)                                  ! Note: need to use untrimmed grid
                chckerr('')
                sol_vec_ser_tot(:,:,id) = sol_vec_dum
            end do
            deallocate(sol_vec_dum)

            norm_factor = max_flux_f/(2*pi)

            ! set up x_plot, y_plot and sol_vec_phase
            allocate(x_plot(grid_sol_trim%n(3),n_mod_tot))
            allocate(y_plot(grid_sol_trim%n(3),n_mod_tot))
            allocate(sol_vec_phase(grid_sol_trim%n(3),n_mod_tot))
            if (use_pol_flux_f) then
                nm_x = 'm'
            else
                nm_x = 'n'
            end if
            do ld = 1,n_mod_tot
                x_plot(:,ld) = grid_sol_trim%r_F
                y_plot(:,ld) = min_nm_x+ld-1
            end do

            ! scale x_plot by normal factor
            x_plot = x_plot/norm_factor

            do id = 0,max_deriv
                ! 1. prepare 2-D plots
                allocate(plot_title(n_mod_tot))
                do ld = 1,n_mod_tot
                    plot_title(ld) = 'EV - midplane ('//nm_x//' = '//&
                        &trim(i2str(min_nm_x+ld-1))//')'
                end do

                ! 2. plot real part at midplane

                ! set up file name of this rank and plot title
                file_name = trim(i2str(x_id))//'_EV_midplane_RE'
                if (id.gt.0) file_name = trim(file_name)//'_D'//trim(i2str(id))

                ! print real amplitude of harmonics of eigenvector at midplane
                call print_ex_2d(plot_title,file_name,&
                    &rp(transpose(sol_vec_ser_tot(:,:,id))),x=x_plot,&
                    &draw=.false.)
                call draw_ex(plot_title,file_name,n_mod_tot,1,&
                    &.false.,draw_ops=['with lines'],ex_plot_style=1)
                call draw_ex(plot_title,file_name,n_mod_tot,1,&
                    &.false.,ex_plot_style=2)

                ! plot in file using decoupled 3D if not too big
                if (n_mod_tot*grid_sol_trim%n(3).le.ex_max_size) then
                    call draw_ex(plot_title,trim(file_name)//'_3D',&
                        &n_mod_tot,3,.false.,draw_ops=['with lines'],&
                        &data_name=file_name,ex_plot_style=1)
                end if

                ! 3. plot imaginary part at midplane

                ! set up file name of this rank and plot title
                file_name = trim(i2str(x_id))//'_EV_midplane_IM'
                if (id.gt.0) file_name = trim(file_name)//'_D'//trim(i2str(id))

                ! print imag amplitude of harmonics of eigenvector at midplane
                call print_ex_2d(plot_title,file_name,&
                    &ip(transpose(sol_vec_ser_tot(:,:,id))),x=x_plot,&
                    &draw=.false.)

                ! plot in file
                call draw_ex(plot_title,file_name,n_mod_tot,1,.false.,&
```

```
                    &draw_ops=['with lines'],ex_plot_style=1)

                ! plot in file using decoupled 3D if not too big
                if (n_mod_tot*grid_sol_trim%n(3).le.ex_max_size) then
                    call draw_ex(plot_title,trim(file_name)//'_3D',&
                        &n_mod_tot,3,.false.,draw_ops=['with lines'],&
                        &data_name=file_name,ex_plot_style=1)
                end if

                ! 4. plot phase at midplane
                ! set up file name of this rank and plot title
                file_name = trim(i2str(x_id))//'_EV_midplane_PH'
                if (id.gt.0) file_name = trim(file_name)//'_D'//trim(i2str(id))

                ! set up phase
                sol_vec_phase = atan2(ip(transpose(sol_vec_ser_tot(:,:,id))),&
                    &rp(transpose(sol_vec_ser_tot(:,:,id))))
                where (sol_vec_phase.lt.0) sol_vec_phase = sol_vec_phase + 2*pi

                ! print imag amplitude of harmonics of eigenvector at midplane
                call print_ex_2d(plot_title,file_name,sol_vec_phase,&
                    &x=x_plot,draw=.false.)

                ! plot in file
                call draw_ex(plot_title,file_name,n_mod_tot,1,&
                    &.false.,draw_ops=['with lines'],ex_plot_style=1)

                ! plot in file using decoupled 3D if not too big
                if (n_mod_tot*grid_sol_trim%n(3).le.ex_max_size) then
                    call draw_ex(plot_title,trim(file_name)//'_3D',&
                        &n_mod_tot,3,.false.,draw_ops=['with lines'],&
                        &data_name=file_name,ex_plot_style=1)
                end if

                ! deallocate variables
                deallocate(plot_title)

                ! 5. prepare 3_D plots
                allocate(plot_title(3))
                plot_title(1) = 'real part'
                plot_title(2) = 'imaginary part'
                plot_title(3) = 'phase'
                file_name = trim(i2str(x_id))//'_EV_midplane'
                if (id.gt.0) file_name = trim(file_name)//'_D'//trim(i2str(id))

                ! 6. plot above in HDf5

                ! plot
                call plot_hdf5(plot_title,trim(file_name),&
                    &reshape([rp(transpose(sol_vec_ser_tot(:,:,id))),&
                    &ip(transpose(sol_vec_ser_tot(:,:,id))),&
                    &sol_vec_phase],[grid_sol_trim%n(3),n_mod_tot,1,3]),&
                    &x=reshape(x_plot,[grid_sol_trim%n(3),n_mod_tot,1,1]),&
                    &y=reshape(y_plot,[grid_sol_trim%n(3),n_mod_tot,1,1]))

                ! deallocate variables
                deallocate(plot_title)
            end do

#if ldebug
            if (debug_plot_harmonics) then
                allocate(plot_title(4))
                allocate(sol_vec_sum(1:grid_sol_trim%n(3),4))
                plot_title(1) = 'EV - outboard midplane - REAL'
                plot_title(2) = 'EV - outboard midplane - IMAG'
                plot_title(3) = 'EV - inboard midplane - REAL'
                plot_title(4) = 'EV - inboard midplane - IMAG'
                do id = 0,max_deriv
                    file_name = 'TEST_harmonics'
                    if (id.gt.0) file_name = trim(file_name)//'_D'//&
                        &trim(i2str(id))
                    sol_vec_sum(:,1) = rp(sum(sol_vec_ser_tot(:,:,id),1))
                    sol_vec_sum(:,2) = ip(sum(sol_vec_ser_tot(:,:,id),1))
                    sol_vec_sum(:,3) = rp(&
                        &sum(sol_vec_ser_tot(1:n_mod_tot:2,:,id),1) - &
                        &sum(sol_vec_ser_tot(2:n_mod_tot:2,:,id),1))
                    sol_vec_sum(:,4) = ip(&
                        &sum(sol_vec_ser_tot(1:n_mod_tot:2,:,id),1) - &
                        &sum(sol_vec_ser_tot(2:n_mod_tot:2,:,id),1))
                    call print_ex_2d(plot_title,file_name,sol_vec_sum,&
                        &x=x_plot(:,1:1),draw=.false.)
                    call draw_ex(plot_title,file_name,4,1,&
                        &.false.,draw_ops=['with lines'],ex_plot_style=1)
                    call draw_ex(plot_title,file_name,4,1,&
                        &.false.,ex_plot_style=2)
                end do
                deallocate(plot_title)
```

```
                deallocate(sol_vec_sum)
            end if
#endif

            ! deallocate variables
            deallocate(x_plot)
            deallocate(y_plot)
            deallocate(sol_vec_phase)

            ! only plot maximum if resonant surfaces found
            if (size(res_surf,1).gt.0) then
                ! set up file name of this rank and plot title
                file_name = trim(i2str(x_id))//'_EV_max'
                allocate(plot_title(2))
                plot_title(1) = 'mode maximum'
                plot_title(2) = 'resonating surface'

                ! set up plot variables
                allocate(x_plot(n_mod_tot,2))
                allocate(y_plot(n_mod_tot,2))

                ! set up maximum of each mode at midplane
                ld_loc = 0
                do ld = 1,n_mod_tot
                    if (maxval(abs(rp(sol_vec_ser_tot(ld,:,0)))).gt.0) then     ! mode resonates somewhere
                        ld_loc = ld_loc + 1
                        x_plot(ld_loc,1) = grid_sol_trim%r_F(&
                            &maxloc(abs(rp(sol_vec_ser_tot(ld,:,0))),1))
                        y_plot(ld_loc,1) = min_nm_x+ld-1
                    end if
                end do
                if (ld_loc.eq.0) then
                    ierr = 1
                    err_msg = 'None of the modes resonates'
                    chckerr(err_msg)
                end if
                x_plot(ld_loc+1:n_mod_tot,1) = x_plot(ld_loc,1)                 ! identical copies of last point for numerical reasons
                y_plot(ld_loc+1:n_mod_tot,1) = y_plot(ld_loc,1)                 ! identical copies of last point for numerical reasons

                ! set up resonating surface for each mode
                x_plot(1:size(res_surf,1),2) = res_surf(:,2)                    ! normal position of resonating mode
                x_plot(size(res_surf,1)+1:n_mod_tot,2) = &
                    &res_surf(size(res_surf,1),2)                               ! identical copies of last point for numerical reasons
                y_plot(1:size(res_surf,1),2) = res_surf(:,1)                    ! total mode index of resonating mode
                y_plot(size(res_surf,1)+1:n_mod_tot,2) = &
                    &res_surf(size(res_surf,1),1)                               ! identical copies of last point for numerical reasons

                ! scale x_plot by normal factor
                x_plot = x_plot/norm_factor

                ! plot the maximum at midplane
                call print_ex_2d(plot_title,file_name,y_plot,x=x_plot,&
                    &draw=.false.)

                ! draw plot in file
                call draw_ex(plot_title,file_name,2,1,.false.,extra_ops=&
                    &'set xrange ['//&
                    &trim(r2str(grid_sol_trim%r_F(1)/norm_factor))//':'//&
                    &trim(r2str(grid_sol_trim%r_F(grid_sol_trim%n(3))/&
                    &norm_factor))//']',ex_plot_style=1)
            end if
        end if

        ! clean up
        call grid_sol_trim%dealloc()
    end function plot_harmonics

    !> Decomposes  the plasma  potential and  kinetic energy  in its  individual
    !! terms for an individual Eigenvalue.
    !!
    !! Use  is made of  variables representing  the potential and  kinetic energy
    !! \cite Weyens3D.
    !!  - \f$E_\text{pot}\f$:
    !!      - normal line bending term:
    !!          \f$\frac{1}{\mu_0} \frac{Q_n^2}{h^{22}}\f$,
    !!      - geodesic line bending term:
    !!          \f$\frac{1}{\mu_0} \mathcal{J}^2 \frac{h^{22}}{g_{33}} Q_g^2\f$,
    !!      - normal ballooning term:
    !!          \f$-2 p' X^2 \kappa_n\f$,
    !!      - geodesic ballooning term:
    !!          \f$-2 p' X U^* \kappa_g\f$,
    !!      - normal kink term:
    !!          \f$-\sigma X^*Q_g\f$,
```

```
    !!      - geodesic kink term:
    !!          \f$\sigma U^*Q_n\f$,
    !!  - \f$E_\text{kin}\f$:
    !!      - normal kinetic term:
    !!          \f$\rho \frac{X^2}{h^{22}}\f$,
    !!      - geodesic kinetic term:
    !!          \f$\rho \mathcal{J}^2 \frac{h^{22}}{g_{33}} U^2\f$.
    !!
    !! The energy terms  are calculated normally on the  sol grid, interpolating
    !! the quantities defined on the eq grid, and angularly in the eq grid.
    !!
    !! Optionally,  the results  can be  plotted  by providing  X, Y  and Z.  By
    !! default, they are instead written to an output file.
    !!
    !! Also, the fraction between potential  and kinetic energy can be returned,
    !! compared with the eigenvalue.
    !!
    !! \return ierr
    integer function decompose_energy(mds_X,mds_sol,grid_eq,grid_X,grid_sol,&
        &eq_1,eq_2,X,sol,vac,X_id,B_aligned,XYZ,E_pot_int,E_kin_int) &
        &result(ierr)

        use grid_utilities, only: trim_grid
        use num_vars, only: no_plots, eq_job_nr, eq_jobs_lims, eq_job_nr

        character(*), parameter :: rout_name = 'decompose_energy'

        ! input / output
        type(modes_type), intent(in) :: mds_x                                   !< general modes variables in pertubation grid
        type(modes_type), intent(in) :: mds_sol                                 !< general modes variables in solution grid
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(grid_type), intent(in) :: grid_x                                   !< perturbation grid
        type(grid_type), intent(in) :: grid_sol                                 !< solution grid
        type(eq_1_type), intent(in) :: eq_1                                     !< flux equilibrium
        type(eq_2_type), intent(in) :: eq_2                                     !< metric equilibrium
        type(x_1_type), intent(in) :: x                                         !< perturbation variables
        type(sol_type), intent(in) :: sol                                       !< solution variables
        type(vac_type), intent(in) :: vac                                       !< vacuum variables
        integer, intent(in) :: x_id                                             !< nr. of Eigenvalue
        logical, intent(in) :: b_aligned                                        !< whether grid is field-aligned
        real(dp), intent(in), optional :: xyz(:,:,:,:)                          !< X, Y and Z for plotting
        complex(dp), intent(inout), optional :: e_pot_int(7)                    !< integrated potential energy
        complex(dp), intent(inout), optional :: e_kin_int(2)                    !< integrated kinetic energy

        ! local variables
        type(grid_type) :: grid_sol_trim                                        ! trimmed sol grid
        integer :: norm_id(2)                                                   ! untrimmed normal indices for trimmed grids
        integer :: kd                                                           ! counter
        integer :: loc_dim(3)                                                   ! local dimension
        integer :: plot_dim(3)                                                  ! total dimensions for plot
        integer :: plot_offset(3)                                               ! local offsets for plot
        logical :: cont_plot                                                    ! continued plot
        complex(dp), allocatable, target :: e_pot(:,:,:,:)                      ! potential energy
        complex(dp), allocatable, target :: e_kin(:,:,:,:)                      ! kinetic energy
        complex(dp) :: e_pot_int_loc(7)                                         ! integrated potential energy for this parallel job
        complex(dp) :: e_kin_int_loc(2)                                         ! integrated kinetic energy for this parallel job
        complex(dp), pointer :: e_pot_trim(:,:,:,:) => null()                   ! trimmed part of E_pot
        complex(dp), pointer :: e_kin_trim(:,:,:,:) => null()                   ! trimmed part of E_kin
        real(dp), allocatable, target :: x_tot(:,:,:,:), y_tot(:,:,:,:), &
            &Z_tot(:,:,:,:)                                                     ! multiple copies of X, Y and Z for collection plot
        real(dp), pointer :: x_tot_trim(:,:,:,:) => null()                      ! trimmed part of X
        real(dp), pointer :: y_tot_trim(:,:,:,:) => null()                      ! trimmed part of Y
        real(dp), pointer :: z_tot_trim(:,:,:,:) => null()                      ! trimmed part of Z
        character(len=max_str_ln), allocatable :: var_names_pot(:)              ! name of potential energy variables
        character(len=max_str_ln), allocatable :: var_names_kin(:)              ! name of kinetic energy variables
        character(len=max_str_ln), allocatable :: var_names(:)                  ! name of other variables that are plot
        character(len=max_str_ln) :: file_name                                  ! name of file
        character(len=max_str_ln) :: description                                ! description

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Calculate energy terms')
        call lvl_ud(1)

        ! calculate for this parallel job
        ierr = calc_e(mds_x,mds_sol,grid_eq,grid_x,grid_sol,eq_1,eq_2,x,sol,&
            &vac,b_aligned,x_id,e_pot,e_kin,e_pot_int_loc,e_kin_int_loc)
        chckerr('')

        ! add to totals if requested
        if (present(e_pot_int)) e_pot_int = e_pot_int + e_pot_int_loc
```

```
        if (present(e_kin_int)) e_kin_int = e_kin_int + e_kin_int_loc

        call lvl_ud(-1)

        ! plot if wanted
        if (present(xyz)) then
            ! bypass plots if no_plots
            if (no_plots) return

            ! user output
            call writo('Preparing plots')
            call lvl_ud(1)

            ! set up continued plot
            cont_plot = eq_job_nr.gt.1

            ! set up potential energy variable names
            allocate(var_names_pot(6))
            var_names_pot(1) = '1. normal line bending term ~ Q_n^2'
            var_names_pot(2) = '2. geodesic line bending term ~ Q_g^2'
            var_names_pot(3) = '3. normal ballooning term ~ -p'' X^2 kappa_n'
            var_names_pot(4) = '4. geodesic ballooning term ~ -p'' X U* kappa_g'
            var_names_pot(5) = '5. normal kink term ~ -sigma X*Q_g'
            var_names_pot(6) = '6. geodesic kink term ~ sigma U*Q_n'

            ! set up kinetic energy variable names
            allocate(var_names_kin(2))
            var_names_kin(1) = '1. normal kinetic term ~ rho X^2'
            var_names_kin(2) = '2. geodesic kinetic term ~ rho U^2'

            ! trim sol grid
            ierr = trim_grid(grid_sol,grid_sol_trim,norm_id)
            chckerr('')

            ! set plot variables
            loc_dim = [grid_eq%n(1),grid_eq%n(2),grid_sol_trim%loc_n_r]
            plot_dim = [grid_eq%n(1),grid_eq%n(2),grid_sol_trim%n(3)]
            plot_offset = [0,0,grid_sol_trim%i_min-1]
            allocate(x_tot(loc_dim(1),loc_dim(2),loc_dim(3),6))
            allocate(y_tot(loc_dim(1),loc_dim(2),loc_dim(3),6))
            allocate(z_tot(loc_dim(1),loc_dim(2),loc_dim(3),6))
            do kd = 1,6
                x_tot(:,:,:,kd) = xyz(:,:,norm_id(1):norm_id(2),1)
                y_tot(:,:,:,kd) = xyz(:,:,norm_id(1):norm_id(2),2)
                z_tot(:,:,:,kd) = xyz(:,:,norm_id(1):norm_id(2),3)
            end do
            allocate(var_names(2))

            ! possibly modify if multiple equilibrium parallel jobs
            if (size(eq_jobs_lims,2).gt.1) then
                plot_dim(1) = eq_jobs_lims(2,size(eq_jobs_lims,2)) - &
                    &eq_jobs_lims(1,1) + 1
                plot_offset(1) = eq_jobs_lims(1,eq_job_nr) - 1
            end if

            ! point to the trimmed versions of energy
            e_pot_trim => e_pot(:,:,norm_id(1):norm_id(2),:)
            e_kin_trim => e_kin(:,:,norm_id(1):norm_id(2),:)

            call lvl_ud(-1)

            ! user output
            call writo('Plot energy terms')
            call lvl_ud(1)

            ! E_kin
            file_name = trim(i2str(x_id))//'_E_kin'
            description = 'Kinetic energy constituents'
            x_tot_trim => x_tot(:,:,:,1:2)
            y_tot_trim => y_tot(:,:,:,1:2)
            z_tot_trim => z_tot(:,:,:,1:2)
            call plot_hdf5(var_names_kin,trim(file_name)//'_RE',rp(e_kin_trim),&
                &tot_dim=[plot_dim,2],loc_offset=[plot_offset,0],&
                &x=x_tot_trim,y=y_tot_trim,z=z_tot_trim,cont_plot=cont_plot,&
                &descr=description)
            call plot_hdf5(var_names_kin,trim(file_name)//'_IM',ip(e_kin_trim),&
                &tot_dim=[plot_dim,2],loc_offset=[plot_offset,0],&
                &x=x_tot_trim,y=y_tot_trim,z=z_tot_trim,cont_plot=cont_plot,&
                &descr=description)
            nullify(x_tot_trim,y_tot_trim,z_tot_trim)

            ! E_pot
            file_name = trim(i2str(x_id))//'_E_pot'
            description = 'Potential energy constituents'
            x_tot_trim => x_tot(:,:,:,1:6)
            y_tot_trim => y_tot(:,:,:,1:6)
            z_tot_trim => z_tot(:,:,:,1:6)
```

```
            call plot_hdf5(var_names_pot,trim(file_name)//'_RE',rp(e_pot_trim),&
                &tot_dim=[plot_dim,6],loc_offset=[plot_offset,0],&
                &x=x_tot_trim,y=y_tot_trim,z=z_tot_trim,cont_plot=cont_plot,&
                &descr=description)
            call plot_hdf5(var_names_pot,trim(file_name)//'_IM',ip(e_pot_trim),&
                &tot_dim=[plot_dim,6],loc_offset=[plot_offset,0],&
                &x=x_tot_trim,y=y_tot_trim,z=z_tot_trim,cont_plot=cont_plot,&
                &descr=description)
            nullify(x_tot_trim,y_tot_trim,z_tot_trim)

            ! E_stab
            var_names(1) = '1. stabilizing term'
            var_names(2) = '2. destabilizing term'
            file_name = trim(i2str(x_id))//'_E_stab'
            description = '(de)stabilizing energy'
            x_tot_trim => x_tot(:,:,:,1:2)
            y_tot_trim => y_tot(:,:,:,1:2)
            z_tot_trim => z_tot(:,:,:,1:2)
            call plot_hdf5(var_names,trim(file_name)//'_RE',rp(reshape(&
                &[sum(e_pot_trim(:,:,:,1:2),4),sum(e_pot_trim(:,:,:,3:6),4)],&
                &[loc_dim(1),loc_dim(2),loc_dim(3),2])),&
                &tot_dim=[plot_dim,2],loc_offset=[plot_offset,0],&
                &x=x_tot_trim,y=y_tot_trim,z=z_tot_trim,&
                &cont_plot=cont_plot,descr=description)
            call plot_hdf5(var_names,trim(file_name)//'_IM',ip(reshape(&
                &[sum(e_pot_trim(:,:,:,1:2),4),sum(e_pot_trim(:,:,:,3:6),4)],&
                &[loc_dim(1),loc_dim(2),loc_dim(3),2])),&
                &tot_dim=[plot_dim,2],loc_offset=[plot_offset,0],&
                &x=x_tot_trim,y=y_tot_trim,z=z_tot_trim,&
                &cont_plot=cont_plot,descr=description)

            ! E
            var_names(1) = '1. potential energy'
            var_names(2) = '2. kinetic energy'
            file_name = trim(i2str(x_id))//'_E'
            description = 'total potential and kinetic energy'
            call plot_hdf5(var_names,trim(file_name)//'_RE',rp(reshape(&
                &[sum(e_pot_trim,4),sum(e_kin_trim,4)],&
                &[loc_dim(1),loc_dim(2),loc_dim(3),2])),&
                &tot_dim=[plot_dim,2],loc_offset=[plot_offset,0],&
                &x=x_tot_trim,y=y_tot_trim,z=z_tot_trim,cont_plot=cont_plot,&
                &descr=description)
            call plot_hdf5(var_names,trim(file_name)//'_IM',ip(reshape(&
                &[sum(e_pot_trim,4),sum(e_kin_trim,4)],&
                &[loc_dim(1),loc_dim(2),loc_dim(3),2])),&
                &tot_dim=[plot_dim,2],loc_offset=[plot_offset,0],&
                &x=x_tot_trim,y=y_tot_trim,z=z_tot_trim,cont_plot=cont_plot,&
                &descr=description)

            call lvl_ud(-1)

            ! clean up
            nullify(e_kin_trim,e_pot_trim)
            nullify(x_tot_trim,y_tot_trim,z_tot_trim)
            call grid_sol_trim%dealloc()
        end if
    end function decompose_energy

    !> Calculate the energy terms in the energy decomposition.
    !!
    !! \note see explanation of routine in decompose_energy().
    !!
    !! \return ierr
    integer function calc_e(mds_X,mds_sol,grid_eq,grid_X,grid_sol,eq_1,eq_2,X,&
        &sol,vac,B_aligned,X_id,E_pot,E_kin,E_pot_int,E_kin_int) result(ierr)

        use num_vars, only: use_pol_flux_f, n_procs, k_style, &
            &norm_disc_prec_eq, norm_disc_prec_x, norm_disc_prec_sol, rank, &
            &eq_job_nr, eq_jobs_lims, x_grid_style
        use eq_vars, only: vac_perm
        use num_utilities, only: c, spline
        use grid_utilities, only: calc_int_vol, trim_grid, untrim_grid
        use grid_vars, only: alpha, n_alpha
        use mpi_utilities, only: get_ser_var
        use sol_utilities, only: calc_xuq

        character(*), parameter :: rout_name = 'calc_E'

        ! input / output
        type(modes_type), intent(in) :: mds_x                                   !< general modes variables in perturbation grid
        type(modes_type), intent(in) :: mds_sol                                 !< general modes variables in solution grid
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(grid_type), intent(in) :: grid_x                                   !< perturbation grid
        type(grid_type), intent(in) :: grid_sol                                 !< and solution grid
        type(eq_1_type), intent(in) :: eq_1                                     !< flux equilibrium
        type(eq_2_type), intent(in) :: eq_2                                     !< metric equilibrium
```

```
        type(x_1_type), intent(in) :: x                                     !< perturbation variables
        type(sol_type), intent(in) :: sol                                   !< solution variables
        type(vac_type), intent(in) :: vac                                   !< vacuum variables
        logical, intent(in) :: b_aligned                                    !< whether grid is field-aligned
        integer, intent(in) :: x_id                                         !< nr. of Eigenvalue
        complex(dp), intent(inout), allocatable :: e_pot(:,:,:,:)           !< potential energy
        complex(dp), intent(inout), allocatable :: e_kin(:,:,:,:)           !< kinetic energy
        complex(dp), intent(inout) :: e_pot_int(7)                          !< integrated potential energy
        complex(dp), intent(inout) :: e_kin_int(2)                          !< integrated kinetic energy

        ! local variables
        integer :: norm_id(2)                                               ! untrimmed normal indices for trimmed grids
        integer :: id, jd, kd                                               ! counters
        integer :: loc_dim(3)                                               ! local dimension
        type(grid_type) :: grid_sol_trim                                    ! trimmed sol grid
        real(dp), allocatable :: h22(:,:,:), g33(:,:,:), j(:,:,:)           ! interpolated h_FD(2,2), g_FD(3,3) and J_FD
        real(dp), allocatable :: kappa_n(:,:,:), kappa_g(:,:,:)             ! interpolated kappa_n and kappa_g
        real(dp), allocatable :: sigma(:,:,:)                               ! interpolated sigma
        real(dp), allocatable :: d2p(:,:,:), rho(:,:,:)                     ! interpolated Dpres_FD, rho
        real(dp), allocatable :: ang_1(:,:,:), ang_2(:,:,:), norm(:)        ! coordinates of grid in which to calculate total energy
        complex(dp), allocatable :: xuq(:,:,:,:)                            ! X, U, Q_n and Q_g
        complex(dp), allocatable :: e_int_tot(:)                            ! integrated potential or kinetic energy of all processes
        character(len=max_str_ln) :: err_msg                                ! error message
#if ldebug
        integer :: plot_dim(3)                                              ! dimensions of plot
        integer :: plot_offset(3)                                           ! local offset of plot
        real(dp), allocatable :: s(:,:,:)                                   ! interpolated S
        complex(dp), allocatable :: du(:,:,:)                               ! D_par U
        complex(dp), allocatable :: du_alt(:,:,:)                           ! DU calculated from U
#endif

        ! set loc_dim
        loc_dim = [grid_eq%n(1:2),grid_sol%loc_n_r]                         ! includes ghost regions of width norm_disc_prec_sol

#if ldebug
        ! set up plot dimensions and offset
        plot_dim = [grid_eq%n(1:2),grid_sol%n(3)]
        plot_offset = [0,0,grid_sol%i_min-1]
#endif

        ! allocate local variables
        allocate(h22(loc_dim(1),loc_dim(2),loc_dim(3)))
        allocate(g33(loc_dim(1),loc_dim(2),loc_dim(3)))
        allocate(j(loc_dim(1),loc_dim(2),loc_dim(3)))
        allocate(kappa_n(loc_dim(1),loc_dim(2),loc_dim(3)))
        allocate(kappa_g(loc_dim(1),loc_dim(2),loc_dim(3)))
        allocate(sigma(loc_dim(1),loc_dim(2),loc_dim(3)))
        allocate(d2p(loc_dim(1),loc_dim(2),loc_dim(3)))
        allocate(rho(loc_dim(1),loc_dim(2),loc_dim(3)))
        allocate(ang_1(loc_dim(1),loc_dim(2),loc_dim(3)))
        allocate(ang_2(loc_dim(1),loc_dim(2),loc_dim(3)))
        allocate(norm(loc_dim(3)))
        allocate(xuq(loc_dim(1),loc_dim(2),loc_dim(3),4))
        allocate(e_pot(loc_dim(1),loc_dim(2),loc_dim(3),6))
        allocate(e_kin(loc_dim(1),loc_dim(2),loc_dim(3),2))
#if ldebug
        if (debug_calc_e .or. debug_du) then
            allocate(du(loc_dim(1),loc_dim(2),loc_dim(3)))
            allocate(s(loc_dim(1),loc_dim(2),loc_dim(3)))

            ! calculate D_par U
            ierr = calc_xuq(mds_x,mds_sol,grid_eq,grid_x,grid_sol,eq_1,eq_2,x,&
                &sol,x_id,2,0._dp,du,deriv=.true.)
            chckerr('')
        end if
#endif

#if lwith_intel
        ! set J to zero to avoid INTEL compiler bug
        j = 0._dp
#endif

        ! interpolate eq->sol
        do jd = 1,grid_eq%n(2)
            do id = 1,grid_eq%n(1)
                ierr = spline(grid_eq%loc_r_F,&
                    &eq_2%h_FD(id,jd,:,c([2,2],.true.),0,0,0),grid_sol%loc_r_F,&
                    &h22(id,jd,:),ord=norm_disc_prec_eq)
                chckerr('')
                ierr = spline(grid_eq%loc_r_F,&
                    &eq_2%g_FD(id,jd,:,c([3,3],.true.),0,0,0),grid_sol%loc_r_F,&
                    &g33(id,jd,:),ord=norm_disc_prec_eq)
                chckerr('')
                ierr = spline(grid_eq%loc_r_F,&
```

```
                    &eq_2%jac_FD(id,jd,:,0,0,0),grid_sol%loc_r_F,&
                    &j(id,jd,:),ord=norm_disc_prec_eq)
                chckerr('')
                ierr = spline(grid_eq%loc_r_F,&
                    &eq_2%kappa_n(id,jd,:),grid_sol%loc_r_F,&
                    &kappa_n(id,jd,:),ord=norm_disc_prec_eq)
                chckerr('')
                ierr = spline(grid_eq%loc_r_F,&
                    &eq_2%kappa_g(id,jd,:),grid_sol%loc_r_F,&
                    &kappa_g(id,jd,:),ord=norm_disc_prec_eq)
                chckerr('')
                ierr = spline(grid_eq%loc_r_F,&
                    &eq_2%sigma(id,jd,:),grid_sol%loc_r_F,&
                    &sigma(id,jd,:),ord=norm_disc_prec_eq)
                chckerr('')
#if ldebug
                if (debug_calc_e) then
                    ierr = spline(grid_eq%loc_r_F,&
                        &eq_2%S(id,jd,:),grid_sol%loc_r_F,&
                        &s(id,jd,:),ord=norm_disc_prec_eq)
                    chckerr('')
                end if
#endif
            end do
        end do
        ierr = spline(grid_eq%loc_r_F,eq_1%pres_FD(:,1),grid_sol%loc_r_F,&
            &d2p(1,1,:),ord=norm_disc_prec_eq)
        chckerr('')
        ierr = spline(grid_eq%loc_r_F,eq_1%rho,grid_sol%loc_r_F,&
            &rho(1,1,:),ord=norm_disc_prec_eq)
        chckerr('')
        do kd = 1,loc_dim(3)
            d2p(:,:,kd) = d2p(1,1,kd)
            rho(:,:,kd) = rho(1,1,kd)
        end do

        ! set angles and norm
        select case (x_grid_style)
            case (1,3)                                                  ! equilibrium or enriched
                ! interpolate X->sol
                if (b_aligned) then
                    if (use_pol_flux_f) then
                        do jd = 1,grid_eq%n(2)
                            do id = 1,grid_eq%n(1)
                                ierr = spline(grid_x%loc_r_F,&
                                    &grid_x%theta_F(id,jd,:),grid_sol%loc_r_F,&
                                    &ang_1(id,jd,:),ord=norm_disc_prec_x)
                                chckerr('')
                            end do
                        end do
                    else
                        do jd = 1,grid_eq%n(2)
                            do id = 1,grid_eq%n(1)
                                ierr = spline(grid_x%loc_r_F,&
                                    &grid_x%zeta_F(id,jd,:),grid_sol%loc_r_F,&
                                    &ang_1(id,jd,:),ord=norm_disc_prec_x)
                                chckerr('')
                            end do
                        end do
                    end if
                    do jd = 1,n_alpha
                        ang_2(:,jd,:) = alpha(jd)
                    end do
                else
                    do jd = 1,grid_eq%n(2)
                        do id = 1,grid_eq%n(1)
                            ierr = spline(grid_x%loc_r_F,&
                                &grid_x%theta_F(id,jd,:),grid_sol%loc_r_F,&
                                &ang_1(id,jd,:),ord=norm_disc_prec_x)
                            chckerr('')
                            ierr = spline(grid_x%loc_r_F,&
                                &grid_x%zeta_F(id,jd,:),grid_sol%loc_r_F,&
                                &ang_2(id,jd,:),ord=norm_disc_prec_x)
                            chckerr('')
                        end do
                    end do
                end if
            case (2)                                                    ! solution
                ! copy
                if (b_aligned) then
                    if (use_pol_flux_f) then
                        ang_1 = grid_x%theta_F
                    else
                        ang_1 = grid_x%zeta_F
                    end if
                    do jd = 1,n_alpha
                        ang_2(:,jd,:) = alpha(jd)
```

```
                    end do
                else
                    ang_1 = grid_x%theta_F
                    ang_2 = grid_x%zeta_F
                end if
        end select
        norm = grid_sol%loc_r_F

        ! calculate X, U, Q_n and Q_g
        do kd = 1,4
            ierr = calc_xuq(mds_x,mds_sol,grid_eq,grid_x,grid_sol,eq_1,eq_2,x,&
                &sol,x_id,kd,0._dp,xuq(:,:,:,kd))
            chckerr('')
        end do

        ! calc kinetic energy
        e_kin(:,:,:,1) = rho/h22*xuq(:,:,:,1)*conjg(xuq(:,:,:,1))
        select case (k_style)
            case (1)                                                        ! normalization of full perpendicular
    component
                e_kin(:,:,:,2) = &
                    &rho*h22*j**2/g33*xuq(:,:,:,2)*conjg(xuq(:,:,:,2))
            case (2)                                                        ! normalization of only normal component
                e_kin(:,:,:,2) = 0._dp
            case default
                err_msg = 'No normalization style associated with '//&
                    &trim(i2str(k_style))
                ierr = 1
                chckerr(err_msg)
        end select

        ! calc potential energy
        e_pot(:,:,:,1) = 1._dp/vac_perm*1._dp/h22*xuq(:,:,:,3)*&
            &conjg(xuq(:,:,:,3))
        e_pot(:,:,:,2) = 1._dp/vac_perm*h22*j**2/g33*xuq(:,:,:,4)*&
            &conjg(xuq(:,:,:,4))
        e_pot(:,:,:,3) = -2*d2p*kappa_n*xuq(:,:,:,1)*conjg(xuq(:,:,:,1))
        e_pot(:,:,:,4) = -2*d2p*kappa_g*xuq(:,:,:,1)*conjg(xuq(:,:,:,2))
        e_pot(:,:,:,5) = -sigma*xuq(:,:,:,4)*conjg(xuq(:,:,:,1))
        e_pot(:,:,:,6) = sigma*xuq(:,:,:,3)*conjg(xuq(:,:,:,2))

#if ldebug
        if (debug_calc_e) then
            call writo('Testing whether the total potential energy is &
                &equal to the alternative form given in [ADD REF]')
            call lvl_ud(1)

            ! alternative formulation for E_pot, always real
            e_pot(:,:,:,3) = (-2*d2p*kappa_n+sigma*s)*&
                &xuq(:,:,:,1)*conjg(xuq(:,:,:,1))
            e_pot(:,:,:,4) = -sigma/j*2*rp(xuq(:,:,:,1)*conjg(du))
            e_pot(:,:,:,5:6) = 0._dp

            call lvl_ud(-1)
        end if

        if (debug_x_norm) then
            call writo('Plotting the squared amplitude of the normal &
                &component')
            call lvl_ud(1)

            call plot_hdf5('X_norm', 'TEST_X_norm_POST_'//trim(i2str(x_id)),&
                &rp(xuq(:,:,:,1)*conjg(xuq(:,:,:,1))),&
                &tot_dim=plot_dim,loc_offset=plot_offset)

            call lvl_ud(-1)
        end if

        if (debug_du .and. b_aligned) then
            call writo('Testing whether DU is indeed the parallel &
                &derivative of U')
            call lvl_ud(1)

            allocate(du_alt(loc_dim(1),loc_dim(2),loc_dim(3)))

            ! derive
            do kd = 1,loc_dim(3)
                do jd = 1,loc_dim(2)
                    ierr = spline(ang_1(:,jd,kd),xuq(:,jd,kd,2),ang_1(:,jd,kd),&
                        &du_alt(:,jd,kd),ord=norm_disc_prec_sol,deriv=1)
                    chckerr('')
                end do
            end do

            ! plot real part
            call plot_hdf5('RE U','TEST_RE_U_'//&
                &trim(i2str(x_id)),rp(xuq(:,:,:,2)),tot_dim=plot_dim,&
```

```
                &loc_offset=plot_offset)
            call plot_diff_hdf5(rp(du),rp(du_alt),&
                &'TEST_RE_DU_'//trim(i2str(x_id)),plot_dim,&
                &plot_offset,descr='To test whether DU is &
                &parallel derivative of U',output_message=.true.)

            ! plot imaginary part
            call plot_hdf5('IM U','TEST_IM_U_'//&
                &trim(i2str(x_id)),ip(xuq(:,:,:,2)),tot_dim=plot_dim,&
                &loc_offset=plot_offset)
            call plot_diff_hdf5(ip(du),ip(du_alt),&
                &'TEST_IM_DU_'//trim(i2str(x_id)),plot_dim,&
                &plot_offset,descr='To test whether DU is &
                &parallel derivative of U',output_message=.true.)

            deallocate(du_alt)

            call lvl_ud(-1)
        end if
#endif

        ! trim sol grid
        ierr = trim_grid(grid_sol,grid_sol_trim,norm_id)
        chckerr('')

        ! add ghost region of width one to the right of the interval
        if (rank.lt.n_procs-1) norm_id(2) = norm_id(2)+1                        ! adjust norm_id as well

        ! integrate energy using ghosted variables
        ierr = calc_int_vol(ang_1(:,:,norm_id(1):norm_id(2)),&
            &ang_2(:,:,norm_id(1):norm_id(2)),norm(norm_id(1):norm_id(2)),&
            &j(:,:,norm_id(1):norm_id(2)),&
            &e_kin(:,:,norm_id(1):norm_id(2),:),e_kin_int)
        chckerr('')
        ierr = calc_int_vol(ang_1(:,:,norm_id(1):norm_id(2)),&
            &ang_2(:,:,norm_id(1):norm_id(2)),norm(norm_id(1):norm_id(2)),&
            &j(:,:,norm_id(1):norm_id(2)),&
            &e_pot(:,:,norm_id(1):norm_id(2),:),e_pot_int(1:6))
        chckerr('')

        ! add vacuum energy if last process and first equilibrium job
        e_pot_int(7) = 0._dp
        write(*,*) '¡¡¡¡¡ NO VACUUM !!!!!'
        if (1.eq.2 .and. rank.eq.n_procs-1 .and. eq_job_nr.eq.size(eq_jobs_lims,2)) then
            do jd = 1,sol%n_mod
                do kd = 1,sol%n_mod
                    e_pot_int(7) = e_pot_int(7) + &
                        &conjg(sol%vec(kd,grid_sol%loc_n_r,x_id)) * &
                        &vac%res(kd,jd) * sol%vec(jd,grid_sol%loc_n_r,x_id)
                end do
            end do
        end if

        ! bundle all processes
        do kd = 1,2
            ierr = get_ser_var([e_kin_int(kd)],e_int_tot,scatter=.true.)
            chckerr('')
            e_kin_int(kd) = sum(e_int_tot)
            deallocate(e_int_tot)
        end do
        do kd = 1,7
            ierr = get_ser_var([e_pot_int(kd)],e_int_tot,scatter=.true.)
            chckerr('')
            e_pot_int(kd) = sum(e_int_tot)
            deallocate(e_int_tot)
        end do

        ! clean up
        call grid_sol_trim%dealloc()
    end function calc_e

    !> Print solution quantities to an output file:
    !!
    !!  - sol:
    !!      - \c val
    !!      - \c vec
    !!
    !! If \c  rich_lvl is provided, \c  "_[R_rich_lvl]" is appended to  the data
    !! name if it is \c >0.
    !!
    !! \return ierr
    integer function print_output_sol(grid,sol,data_name,rich_lvl,&
        &remove_previous_arrs) result(ierr)

        use num_vars, only: pb3d_name, sol_n_procs, rank
        use hdf5_ops, only: print_hdf5_arrs
        use hdf5_vars, only: dealloc_var_1d, var_1d_type, &
```

```
            &max_dim_var_1d
        use grid_utilities, only: trim_grid

        character(*), parameter :: rout_name = 'print_output_sol'

        ! input / output
        type(grid_type), intent(in) :: grid                                     !< solution grid variables
        type(sol_type), intent(in) :: sol                                       !< solution variables
        character(len=*), intent(in) :: data_name                               !< name under which to store
        integer, intent(in), optional :: rich_lvl                               !< Richardson level to print
        logical, intent(in), optional :: remove_previous_arrs                   !< remove previous variables if present

        ! local variables
        type(var_1d_type), allocatable, target :: sol_1d(:)                     ! 1D equivalent of eq. variables
        type(var_1d_type), pointer :: sol_1d_loc => null()                      ! local element in sol_1D
        type(grid_type) :: grid_trim                                            ! trimmed solution grid
        integer :: id                                                           ! counter

        ! initialize ierr
        ierr = 0

        ! only write to output file if at least one solution
        if (size(sol%val).gt.0) then
            ! user output
            call writo('Write solution variables to output file')
            call lvl_ud(1)

            ! trim grids
            ierr = trim_grid(grid,grid_trim)
            chckerr('')

            ! Set up the 1D equivalents of the solution variables
            allocate(sol_1d(max_dim_var_1d))

            id = 1

            ! RE_sol_val
            sol_1d_loc => sol_1d(id); id = id+1
            sol_1d_loc%var_name = 'RE_sol_val'
            allocate(sol_1d_loc%tot_i_min(1),sol_1d_loc%tot_i_max(1))
            allocate(sol_1d_loc%loc_i_min(1),sol_1d_loc%loc_i_max(1))
            sol_1d_loc%tot_i_min = [1]
            sol_1d_loc%tot_i_max = [size(sol%val)]
            sol_1d_loc%loc_i_min = [1]
            sol_1d_loc%loc_i_max = [size(sol%val)]
            allocate(sol_1d_loc%p(size(sol%val)))
            sol_1d_loc%p = rp(sol%val)

            ! IM_sol_val
            sol_1d_loc => sol_1d(id); id = id+1
            sol_1d_loc%var_name = 'IM_sol_val'
            allocate(sol_1d_loc%tot_i_min(1),sol_1d_loc%tot_i_max(1))
            allocate(sol_1d_loc%loc_i_min(1),sol_1d_loc%loc_i_max(1))
            sol_1d_loc%tot_i_min = [1]
            sol_1d_loc%tot_i_max = [size(sol%val)]
            sol_1d_loc%loc_i_min = [1]
            sol_1d_loc%loc_i_max = [size(sol%val)]
            allocate(sol_1d_loc%p(size(sol%val)))
            sol_1d_loc%p = ip(sol%val)

            ! RE_sol_vec
            sol_1d_loc => sol_1d(id); id = id+1
            sol_1d_loc%var_name = 'RE_sol_vec'
            allocate(sol_1d_loc%tot_i_min(3),sol_1d_loc%tot_i_max(3))
            allocate(sol_1d_loc%loc_i_min(3),sol_1d_loc%loc_i_max(3))
            sol_1d_loc%loc_i_min = [1,grid_trim%i_min,1]
            sol_1d_loc%loc_i_max = [sol%n_mod,grid_trim%i_max,size(sol%vec,3)]
            sol_1d_loc%tot_i_min = [1,1,1]
            sol_1d_loc%tot_i_max = [sol%n_mod,grid_trim%n(3),size(sol%vec,3)]
            allocate(sol_1d_loc%p(size(sol%vec)))
            sol_1d_loc%p = reshape(rp(sol%vec),[size(sol%vec)])

            ! IM_sol_vec
            sol_1d_loc => sol_1d(id); id = id+1
            sol_1d_loc%var_name = 'IM_sol_vec'
            allocate(sol_1d_loc%tot_i_min(3),sol_1d_loc%tot_i_max(3))
            allocate(sol_1d_loc%loc_i_min(3),sol_1d_loc%loc_i_max(3))
            sol_1d_loc%loc_i_min = [1,grid_trim%i_min,1]
            sol_1d_loc%loc_i_max = [sol%n_mod,grid_trim%i_max,size(sol%vec,3)]
            sol_1d_loc%tot_i_min = [1,1,1]
            sol_1d_loc%tot_i_max = [sol%n_mod,grid_trim%n(3),size(sol%vec,3)]
            allocate(sol_1d_loc%p(size(sol%vec)))
            sol_1d_loc%p = reshape(ip(sol%vec),[size(sol%vec)])

            ! write
            ! Note: if processes that are not used  in the solve do not have the
            ! solution  vector and  should therefore  not be  used to  write the
```

```
            ! output.
            if (sol_n_procs.gt.1 .or. rank.eq.0) then
                ierr = print_hdf5_arrs(sol_1d(1:id-1),pb3d_name,&
                    &trim(data_name),rich_lvl=rich_lvl,&
                    &remove_previous_arrs=remove_previous_arrs)
                chckerr('')
            end if

            ! clean up
            call grid_trim%dealloc()
            call dealloc_var_1d(sol_1d)
            nullify(sol_1d_loc)

            ! user output
            call lvl_ud(-1)
        else
            ! user output
            call writo('No solutions to write')
        end if
    end function print_output_sol
end module sol_ops
```

D.78 /Users/toonweyens/Code/PB3D/Modules/sol_utilities.f90 File Reference

Interfaces and Types

- interface sol_utilities::calc_xuq

  Calculates the normal ($\cdot_n$) or geodesic ($\cdot_g$) components of the plasma perturbation $\vec{\xi}$ or the magnetic perturbation $\vec{Q} = \nabla \times \left(\vec{x} \times \vec{B}\right)$.

Modules

- module sol_utilities

  Numerical utilities related to the solution vectors.

Functions/Subroutines

- integer function, public sol_utilities::calc_tot_sol_vec (mds, grid_sol, sol_vec_loc, sol_vec_tot, deriv)

  Calculate the total version of the solution vector from the local version.
- integer function, public sol_utilities::calc_loc_sol_vec (mds, i_min, sol_vec_tot, sol_vec_loc)

  Calculate the local version of the solution vector from the total version.
- integer function, public sol_utilities::interp_v_spline (v_i, v_o, r_i, r_o, extrap, ivs_stat)

  Interpolation for a quantity V using splines.

Variables

- logical, public sol_utilities::debug_calc_xuq_arr = .false.

  plot debug information for calc_XUQ_arr
- logical, public sol_utilities::debug_interp_v_spline = .false.

  debug information for interp_v_spline

D.79 sol_utilities.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Numerical utilities related to the solution vectors.
!-----------------------------------------------------------------------------!
module sol_utilities
#include <PB3D_macros.h>
#include <wrappers.h>
    use str_utilities
    use output_ops
    use messages
    use num_vars, only: dp, iu, max_str_ln, pi
    use grid_vars, only: grid_type
    use eq_vars, only: eq_1_type, eq_2_type
    use x_vars, only: x_1_type, modes_type
    use sol_vars, only: sol_type

    implicit none
    private
    public calc_xuq, calc_tot_sol_vec, calc_loc_sol_vec, interp_v_spline
#if ldebug
    public debug_calc_xuq_arr, debug_interp_v_spline
#endif

    ! global variables
#if ldebug
    logical :: debug_calc_xuq_arr = .false.                                  !< plot debug information for calc_XUQ_arr \ldebug
    logical :: debug_interp_v_spline = .false.                               !< debug information for interp_v_spline \ldebug
#endif

    ! interfaces

    !> \public Calculates the normal (\f$\cdot_n\f$) or geodesic (\f$\cdot_g\f$)
    !! components of the plasma perturbation \f$\vec{\xi}\f$ or the
    !! magnetic perturbation \f$\vec{Q} = \nabla \times \left(\vec{x} \times
    !! \vec{B}\right)\f$.
    !!
    !! Which variables are plot depends on \c XUQ_style \cite Weyens3D :
    !!  - \c XUQ_style = 1: \f$X\f$ (supports parallel derivative)
    !!  - \c XUQ_style = 2: \f$U\f$ (supports parallel derivative)
    !!  - \c XUQ_style = 3: \f$Q_n\f$
    !!  - \c XUQ_style = 4: \f$Q_g\f$
    !!
    !! \f$X\f$ and \f$U\f$ support the calculation of the parallel derivative.
    !!
    !! For \f$Q_n\f$ and \f$Q_g\f$, the metric variables have to be provided as
    !! well.
    !!
    !! The perturbation grid is assumed to have the same angular coordinates as
    !! the equilibrium grid, and the normal coordinates correspond to either the
    !! equilibrium grid (\c X_grid_style 1), the solution grid (\c X_grid_style
    !! 2) or the enriched equilibrium grid (\c X_grid_style 3).
    !!
    !! The output grid, furthermore, has the angular part corresponding to the
    !! equilibrium grid, and the normal part to the solution grid.
    !!
    !! \return ierr
    interface calc_xuq
        !> \public
        module procedure calc_xuq_ind
        !> \public
        module procedure calc_xuq_arr
    end interface

contains
    !> \private (time) array version
    integer function calc_xuq_arr(mds_X,mds_sol,grid_eq,grid_X,grid_sol,eq_1,&
        &eq_2,X,sol,X_id,XUQ_style,time,XUQ,deriv) result(ierr)

        use num_vars, only: use_pol_flux_f, norm_disc_prec_sol, x_grid_style
        use num_utilities, only: con2dis, spline
        use x_utilities, only: trim_modes
#if ldebug
        use eq_vars, only: max_flux_f
        use num_utilities, only: calc_int
#endif

        character(*), parameter :: rout_name = 'calc_XUQ_arr'

        ! input / output
        type(modes_type), intent(in), target :: mds_x                        !< general modes variables in perturbation grid
```

```
        type(modes_type), intent(in), target :: mds_sol                 !< general modes variables in solution grid
        type(grid_type), intent(in) :: grid_eq                          !< equilibrium grid
        type(grid_type), intent(in) :: grid_x                           !< perturbation grid
        type(grid_type), intent(in) :: grid_sol                         !< solution grid
        type(eq_1_type), intent(in) :: eq_1                             !< flux equilibrium
        type(eq_2_type), intent(in) :: eq_2                             !< metric equilibrium
        type(x_1_type), intent(in), target :: x                         !< perturbation variables
        type(sol_type), intent(in) :: sol                               !< solution variables
        integer, intent(in) :: x_id                                     !< nr. of Eigenvalue
        integer, intent(in) :: xuq_style                                !< whether to calculate \f$X\f$, \f$U\f$, \f$Q_n\f$ or \f$Q_g\f$
        real(dp), intent(in) :: time(:)                                 !< time range
        complex(dp), intent(inout) :: xuq(:,:,:,:)                      !< \f$X\f$, \f$U\f$, \f$Q_n\f$ or \f$Q_g\f$
        logical, intent(in), optional :: deriv                          !< return parallel derivative

        ! local variables
        integer :: m                                                    ! secondary mode number
        integer :: kd_tot                                               ! kd in total grid
        integer :: ind_x                                                ! index of mode in X tables
        integer :: ind_sol                                              ! index of mode in sol tables
        integer :: n_mod_tot                                            ! total number of modes
        integer :: n_t                                                  ! number of time points
        integer :: n_loc, m_loc                                         ! local mode numbers
        integer :: id, jd, kd                                           ! counters
        integer :: xuq_dims(4)                                          ! dimensions of XUQ
        integer :: kdl_x(2), kdl_sol(2)                                 ! limits on normal index for a mode in X and sol grids
        integer :: id_lim_x(2), id_lim_sol(2)                           ! limits on total modes
        integer, pointer :: sec_x_loc(:,:), sec_sol_loc(:,:)            ! pointers to secondary mode variables
        real(dp), allocatable :: expon(:,:,:)                           ! exponent in solution grid
        real(dp), allocatable :: par_fac(:)                             ! multiplicative factor due to parallel derivative, without iu, in perturbation grid
        real(dp), allocatable :: jq(:)                                  ! iota or q, interpolated at perturbation grid
        real(dp), allocatable :: s(:,:,:), inv_j(:,:,:)                 ! S and 1/J, interpolated at perturbation grid
        real(dp), pointer :: r_x_loc(:), r_sol_loc(:)                   ! local r_X and r_sol for a mode
        real(dp), pointer :: theta_f(:,:,:), zeta_f(:,:,:)              ! angles in solution normal grid
        complex(dp) :: sqrt_sol_val_norm                                ! normalized sqrt(sol_val)
        complex(dp), allocatable :: fac_0(:,:,:), fac_1(:,:,:)          ! factor to be multiplied with X and DX in perturbation grid
        complex(dp), allocatable :: xuq_loc(:,:)                        ! complex XUQ without time at a normal point
        complex(dp), allocatable :: sol_vec(:,:)                        ! total solution vector in solution grid
        complex(dp), allocatable :: sol_vec_tot(:,:,:)                  ! total solution vector in solution grid and derivatives
        complex(dp), allocatable :: dsol_vec(:,:)                       ! normal derivative of sol_vec in solution grid
        complex(dp), pointer :: u0(:,:,:), u1(:,:,:)                    ! U0 and U1 or parallel derivative
        logical :: extrap = .true.                                      ! whether extrapolation is used
        logical :: deriv_loc                                            ! local copy of deriv
        character(len=max_str_ln) :: err_msg                            ! error message

#if ldebug
        real(dp), allocatable :: sol_vec_alt(:)                         ! sol_vec calculated from Dsol_vec in solution grid
#endif

        ! initialize ierr
        ierr = 0

        ! set up dimensions
        n_t = size(time)
        xuq_dims = shape(xuq)

        ! set up local copy of deriv
        deriv_loc = .false.
        if (present(deriv)) deriv_loc = deriv

        ! tests
        if (grid_eq%n(1).ne.grid_x%n(1) .or. grid_eq%n(2).ne.grid_x%n(2)) then
            ierr = 1
            err_msg = 'Grids need to be compatible'
            chckerr(err_msg)
        end if
        if (grid_eq%n(1).ne.xuq_dims(1) .or. grid_eq%n(2).ne.xuq_dims(2) .or. &
            &grid_sol%loc_n_r.ne.xuq_dims(3) .or. n_t.ne.xuq_dims(4)) then
            ierr = 1
            err_msg = 'XUQ needs to have the correct dimensions'
            chckerr(err_msg)
        end if

        ! limit input modes to output modes
        ! (X grid should comprise sol grid)
        ierr = trim_modes(mds_x,mds_sol,id_lim_x,id_lim_sol)
        chckerr('')
        sec_x_loc => mds_x%sec(id_lim_x(1):id_lim_x(2),:)
        sec_sol_loc => mds_sol%sec(id_lim_sol(1):id_lim_sol(2),:)
```

```fortran

        n_mod_tot = size(sec_x_loc,1)

        if (size(sec_x_loc,1).ne.size(sec_sol_loc,1)) then
            ierr = 1
            err_msg = 'modes X and sol have inconsistent sizes'
            chckerr(err_msg)
        end if

        ! set up normalized sqrt(sol_val)
        sqrt_sol_val_norm = sqrt(sol%val(x_id))
        if (ip(sqrt_sol_val_norm).gt.0) sqrt_sol_val_norm = - sqrt_sol_val_norm ! exploding solution, not the decaying one
        sqrt_sol_val_norm = sqrt_sol_val_norm / abs(sqrt_sol_val_norm)          ! normalize

        ! allocate helper variables
        allocate(jq(xuq_dims(3)))
        allocate(s(xuq_dims(1),xuq_dims(2),xuq_dims(3)))
        if ((xuq_style.eq.3 .or. xuq_style.eq.4)) &
            &allocate(inv_j(xuq_dims(1),xuq_dims(2),xuq_dims(3)))
        allocate(xuq_loc(xuq_dims(1),xuq_dims(2)))
        allocate(dsol_vec(sol%n_mod,xuq_dims(3)))

        ! interpolate eq->sol
        if (use_pol_flux_f) then
            ierr = spline(grid_eq%loc_r_F,eq_1%q_saf_FD(:,0),&
                &grid_sol%loc_r_F,jq,ord=norm_disc_prec_sol)
            chckerr('')
        else
            ierr = spline(grid_eq%loc_r_F,eq_1%rot_t_FD(:,0),&
                &grid_sol%loc_r_F,jq,ord=norm_disc_prec_sol)
            chckerr('')
        end if
        do jd = 1,xuq_dims(2)
            do id = 1,xuq_dims(1)
                ierr = spline(grid_eq%loc_r_F,eq_2%S(id,jd,:),&
                    &grid_sol%loc_r_F,s(id,jd,:),ord=norm_disc_prec_sol)
                chckerr('')
                if ((xuq_style.eq.3 .or. xuq_style.eq.4)) then
                    ierr = spline(grid_eq%loc_r_F,&
                        &1._dp/eq_2%jac_FD(id,jd,:,0,0,0),grid_sol%loc_r_F,&
                        &inv_j(id,jd,:),ord=norm_disc_prec_sol)
                    chckerr('')
                end if
            end do
        end do

        ! initialize XUQ
        xuq = 0._dp

        ! set up normal derivative of X vec
        ! Note: In X_style 2 (fast), the mode  indices vary as a function as the
        ! normal coordinate.  All of  these contiguous  ranges are  tabulated in
        ! variable 'sec'.
        if (xuq_style.eq.2 .or. xuq_style.eq.4) then
            ! set up variables
            allocate(sol_vec_tot(n_mod_tot,grid_sol%loc_n_r,0:1))
            sol_vec_tot = 0._dp

            ! convert to total solution vector and derivate
            do id = 0,1
                ierr = calc_tot_sol_vec(mds_sol,grid_sol,sol%vec(:,:,x_id),&
                    &sol_vec,deriv=id)
                chckerr('')
                sol_vec_tot(:,:,id) = sol_vec
            end do

#if ldebug
            if (debug_calc_xuq_arr) then
                do m = 1,n_mod_tot
                    call writo('For mode '//trim(i2str(m))//', testing &
                        &whether Dsol_vec is correct by comparing its integral &
                        &with original sol_vec')
                    kdl_sol = sec_sol_loc(m,2:3)
                    allocate(sol_vec_alt(1:kdl_sol(2)-kdl_sol(1)+1))
                    sol_vec_alt = 0._dp
                    ierr = calc_int(rp(sol_vec_tot(m,kdl_sol(1):kdl_sol(2),1)),&
                        &grid_sol%loc_r_F(kdl_sol(1):kdl_sol(2)),sol_vec_alt)
                    chckerr('')
                    call print_ex_2d(['TEST_RE_Dsol_vec'],'',reshape(&
                        &[rp(sol_vec_tot(m,kdl_sol(1):kdl_sol(2),0)),&
                        &sol_vec_alt],[kdl_sol(2)-kdl_sol(1)+1,2]),x=reshape(&
                        &grid_sol%loc_r_F(kdl_sol(1):kdl_sol(2)),&
                        &[kdl_sol(2)-kdl_sol(1)+1,1])/max_flux_f*2*pi)
                    ierr = calc_int(ip(sol_vec_tot(m,kdl_sol(1):kdl_sol(2),1)),&
                        &grid_sol%loc_r_F(kdl_sol(1):kdl_sol(2)),sol_vec_alt)
                    chckerr('')
                    call print_ex_2d(['TEST_IM_Dsol_vec'],'',reshape(&
```

```
                        &[ip(sol_vec_tot(m,kdl_sol(1):kdl_sol(2),0)),&
                        &sol_vec_alt],[kdl_sol(2)-kdl_sol(1)+1,2]),x=reshape(&
                        &grid_sol%loc_r_F(kdl_sol(1):kdl_sol(2)),&
                        &[kdl_sol(2)-kdl_sol(1)+1,1])/max_flux_f*2*pi)
                    deallocate(sol_vec_alt)
                end do
            end if
#endif

            ! convert back to local solution vector
            ierr = calc_loc_sol_vec(mds_sol,grid_sol%i_min,sol_vec_tot(:,:,1),&
                &dsol_vec)
            chckerr('')

            ! clean up
            deallocate(sol_vec,sol_vec_tot)
        else
            dsol_vec = 0._dp
        end if

        ! interpolate or point angles
        select case (x_grid_style)
            case (1,3)                                                      ! equilibrium or enriched
                allocate(theta_f(xuq_dims(1),xuq_dims(2),xuq_dims(3)))
                allocate(zeta_f(xuq_dims(1),xuq_dims(2),xuq_dims(3)))
                do jd = 1,xuq_dims(2)
                    do id = 1,xuq_dims(1)
                        ierr = spline(grid_eq%loc_r_F,grid_eq%theta_F(id,jd,:),&
                            &grid_sol%loc_r_F,theta_f(id,jd,:))
                        chckerr('')
                        ierr = spline(grid_eq%loc_r_F,grid_eq%zeta_F(id,jd,:),&
                            &grid_sol%loc_r_F,zeta_f(id,jd,:))
                        chckerr('')
                    end do
                end do
            case (2)                                                        ! solution
                theta_f => grid_x%theta_F
                zeta_f => grid_x%zeta_F
        end select

        ! iterate over all secondary modes
        do m = 1,n_mod_tot
            ! test
            if (sec_x_loc(m,1).ne.sec_sol_loc(m,1)) then
                ierr = 1
                err_msg = 'For mode '//trim(i2str(m))//&
                    &', no consistency for modes in grid_X and grid_sol'
                chckerr(err_msg)
            end if

            ! set up and normal limits
            kdl_x = sec_x_loc(m,2:3)
            kdl_sol = sec_sol_loc(m,2:3)

            ! limit to grid ranges
            kdl_x(1) = max(kdl_x(1),grid_x%i_min)
            kdl_x(2) = min(kdl_x(2),grid_x%i_max)
            kdl_sol(1) = max(kdl_sol(1),grid_sol%i_min)
            kdl_sol(2) = min(kdl_sol(2),grid_sol%i_max)

            ! convert limits to local
            kdl_x = kdl_x - grid_x%i_min + 1
            kdl_sol = kdl_sol - grid_sol%i_min + 1

            ! cycle if mode does not exist anywhere
            if (kdl_x(1).gt.kdl_x(2) .or. kdl_sol(1).gt.kdl_sol(2)) cycle

            ! get perturbation and solution normal ranges for this mode number
            r_x_loc => grid_x%loc_r_F(kdl_x(1):kdl_x(2))
            r_sol_loc => grid_sol%loc_r_F(kdl_sol(1):kdl_sol(2))

            ! set up table indices
            ind_x = sec_x_loc(m,4)
            ind_sol = sec_sol_loc(m,4)

            ! Set up multiplicative factors fac_0 and fac_1 which are used in
            ! XUQ = fac_0*X + fac_1*DX with fac_0 and fac_1:
            !   1. fac_0 = 1 (i(nq-m) or i(n-iota m) for deriv)
            !      fac_1 = 0
            !   2. fac_0 = U_0 (DU_0 for deriv)
            !      fac_1 = U_1 (DU_1 for deriv)
            !   3. fac_0 = i(nq-m)/J or i(n-iota m)/J (no deriv)
            !      fac_1 = 0
            !   4. fac_0 = DU_0/J - S (no deriv)
            !      fac_1 = DU_1/J
            ! where par_fac = nq-m or n-iota m.
            ! These quantities are then multiplied by the exponent for each mode
```

```
            ! number.
            allocate(expon(xuq_dims(1),xuq_dims(2),kdl_sol(1):kdl_sol(2)))      ! normal subset of grid_sol%loc_r_F
            allocate(fac_0(xuq_dims(1),xuq_dims(2),kdl_sol(1):kdl_sol(2)))      ! normal subset of grid_sol%loc_r_F
            allocate(fac_1(xuq_dims(1),xuq_dims(2),kdl_sol(1):kdl_sol(2)))      ! normal subset of grid_sol%loc_r_F
            allocate(par_fac(kdl_sol(1):kdl_sol(2)))                            ! normal subset of grid_sol%loc_r_F

            ! calculate parallel multiplicative factor par_fac and exponent
            do kd = kdl_sol(1),kdl_sol(2)
                ! set up local mode numbers
                kd_tot = kd + grid_sol%i_min - 1
                n_loc = mds_sol%n(kd_tot,ind_sol)
                m_loc = mds_sol%m(kd_tot,ind_sol)
#if ldebug
                if (use_pol_flux_f .and. (m_loc.ne.sec_x_loc(m,1)) .or. &
                    &(.not.use_pol_flux_f .and. (n_loc.ne.sec_x_loc(m,1)))) then
                    ierr = 1
                    err_msg = 'mode numbers not consistent'
                    chckerr(err_msg)
                end if
#endif

                if (use_pol_flux_f) then
                    par_fac(kd) = n_loc*jq(kd) - m_loc
                else
                    par_fac(kd) = n_loc - m_loc*jq(kd)
                end if
                expon(:,:,kd) = n_loc*zeta_f(:,:,kd) - &
                    &m_loc*theta_f(:,:,kd)
            end do

            ! set up multiplicative factors in solution grid
            select case (xuq_style)
                case (1)                                                        ! calculating X
                    if (deriv_loc) then                                         ! parallel derivative
                        do kd = kdl_sol(1),kdl_sol(2)
                            fac_0(:,:,kd) = iu*par_fac(kd)
                        end do
                    else
                        fac_0 = 1._dp
                    end if
                    fac_1 = 0._dp
                case (2)                                                        ! calculating U
                    ! interpolate X-> sol or point
                    select case (x_grid_style)
                        case (1,3)                                              ! equilibrium or enriched
                            allocate(u0(xuq_dims(1),xuq_dims(2),&
                                &kdl_sol(1):kdl_sol(2)))
                            allocate(u1(xuq_dims(1),xuq_dims(2),&
                                &kdl_sol(1):kdl_sol(2)))
                            if (deriv_loc) then                                 ! parallel derivative
                                ierr = interp_v_spline(&
                                    &x%DU_0(:,:,kdl_x(1):kdl_x(2),ind_x),u0,&
                                    &r_x_loc,r_sol_loc,extrap)
                                chckerr('')
                                ierr = interp_v_spline(&
                                    &x%DU_1(:,:,kdl_x(1):kdl_x(2),ind_x),u1,&
                                    &r_x_loc,r_sol_loc,extrap)
                                chckerr('')
                            else
                                ierr = interp_v_spline(&
                                    &x%U_0(:,:,kdl_x(1):kdl_x(2),ind_x),u0,&
                                    &r_x_loc,r_sol_loc,extrap)
                                chckerr('')
                                ierr = interp_v_spline(&
                                    &x%U_1(:,:,kdl_x(1):kdl_x(2),ind_x),u1,&
                                    &r_x_loc,r_sol_loc,extrap)
                                chckerr('')
                            end if
                        case (2)                                                ! solution
                            if (deriv_loc) then                                 ! parallel derivative
                                u0 => x%DU_0(:,:,kdl_x(1):kdl_x(2),ind_x)
                                u1 => x%DU_1(:,:,kdl_x(1):kdl_x(2),ind_x)
                            else
                                u0 => x%U_0(:,:,kdl_x(1):kdl_x(2),ind_x)
                                u1 => x%U_1(:,:,kdl_x(1):kdl_x(2),ind_x)
                            end if
                    end select

                    ! assign
                    fac_0 = u0
                    fac_1 = u1

                    ! clean up
                    select case (x_grid_style)
                        case (1,3)                                              ! equilibrium or enriched
                            deallocate(u0,u1)
                        case (2)                                                ! solution
```

```
                        ! do nothing
                    end select
                case (3)                                                    ! calculating Qn
                    if (deriv_loc) then                                     ! parallel derivative
                        ierr = 1
                        err_msg = 'For XUQ_style = '//trim(i2str(xuq_style))//&
                            &' calculation of parallel derivative not supported'
                        chckerr('')
                    else
                        do kd = kdl_sol(1),kdl_sol(2)
                            fac_0(:,:,kd) = iu*par_fac(kd)*inv_j(:,:,kd)
                        end do
                    end if
                    fac_1 = 0._dp
                case (4)                                                    ! calculating Qg
                    ! interpolate X-> sol or point
                    err_msg = 'For XUQ_style = '//trim(i2str(xuq_style))//&
                        &' calculation of parallel derivative not supported'
                    chckerr('')
                    select case (x_grid_style)
                        case (1,3)                                          ! equilibrium or enriched
                            allocate(u0(xuq_dims(1),xuq_dims(2),&
                                &kdl_sol(1):kdl_sol(2)))
                            allocate(u1(xuq_dims(1),xuq_dims(2),&
                                &kdl_sol(1):kdl_sol(2)))
                            if (deriv_loc) then                             ! parallel derivative
                                ierr = 1
                                chckerr(err_msg)
                            else
                                ierr = interp_v_spline(&
                                    &x%DU_0(:,:,kdl_x(1):kdl_x(2),ind_x),u0,&
                                    &r_x_loc,r_sol_loc,extrap)
                                chckerr('')
                                ierr = interp_v_spline(&
                                    &x%DU_1(:,:,kdl_x(1):kdl_x(2),ind_x),u1,&
                                    &r_x_loc,r_sol_loc,extrap)
                                chckerr('')
                            end if
                        case (2)                                            ! solution
                            if (deriv_loc) then                             ! parallel derivative
                                ierr = 1
                                chckerr(err_msg)
                            else
                                u0 => x%DU_0(:,:,kdl_x(1):kdl_x(2),ind_x)
                                u1 => x%DU_1(:,:,kdl_x(1):kdl_x(2),ind_x)
                            end if
                    end select

                    ! assign
                    fac_0 = u0*inv_j(:,:,kdl_sol(1):kdl_sol(2)) - &
                        &s(:,:,kdl_sol(1):kdl_sol(2))
                    fac_1 = u1*inv_j(:,:,kdl_sol(1):kdl_sol(2))

                    ! clean up
                    select case (x_grid_style)
                        case (1,3)                                          ! equilibrium or enriched
                            deallocate(u0,u1)
                        case (2)                                            ! solution
                            ! do nothing
                    end select
                case default
                    err_msg = 'No XUQ style associated with '//&
                        &trim(i2str(xuq_style))
                    ierr = 1
                    chckerr(err_msg)
            end select

            ! iterate over all normal points in solution modes variables
            do kd = kdl_sol(1),kdl_sol(2)
                ! set up loc complex XUQ without time at this normal point
                xuq_loc = exp(iu*expon(:,:,kd)) * &
                    &(sol%vec(ind_sol,kd,x_id)*fac_0(:,:,kd) + &
                    &dsol_vec(ind_sol,kd)*fac_1(:,:,kd))

                ! iterate over time steps
                do id = 1,n_t
                    ! add current mode
                    xuq(:,:,kd,id) = xuq(:,:,kd,id) + xuq_loc * &
                        &exp(iu*sqrt_sol_val_norm*time(id)*2*pi)
                end do
            end do

            ! clean up
            deallocate(fac_0,fac_1,expon,par_fac)
        end do

        ! clean up
```

```
        select case (x_grid_style)
            case (1,3)                                                  ! equilibrium or enriched
                deallocate(theta_f,zeta_f)
            case (2)                                                    ! solution
                ! do nothing
        end select
        nullify(r_x_loc,r_sol_loc)
        nullify(theta_f,zeta_f)
        nullify(u0,u1)
        nullify(sec_x_loc,sec_sol_loc)
    end function calc_xuq_arr
    !> \private (time) individual version
    integer function calc_xuq_ind(mds_X,mds_sol,grid_eq,grid_X,grid_sol,eq_1,&
        &eq_2,X,sol,X_id,XUQ_style,time,XUQ,deriv) result(ierr)

        character(*), parameter :: rout_name = 'calc_XUQ_ind'

        ! input / output
        type(modes_type), intent(in) :: mds_x                           !< general modes variables in perturbation grid
        type(modes_type), intent(in) :: mds_sol                         !< general modes variables in solution grid
        type(grid_type), intent(in) :: grid_eq                          !< equilibrium grid
        type(grid_type), intent(in) :: grid_x                           !< perturbation grid
        type(grid_type), intent(in) :: grid_sol                         !< solution grid
        type(eq_1_type), intent(in) :: eq_1                             !< flux equilibrium
        type(eq_2_type), intent(in) :: eq_2                             !< metric equilibrium
        type(x_1_type), intent(in) :: x                                 !< perturbation variables
        type(sol_type), intent(in) :: sol                               !< solution variables
        integer, intent(in) :: x_id                                     !< nr. of Eigenvalue
        integer, intent(in) :: xuq_style                                !< whether to calculate X, U, Qn or Qg
        real(dp), intent(in) :: time                                    !< time
        complex(dp), intent(inout) :: xuq(:,:,:)                        !< normal component of perturbation
        logical, intent(in), optional :: deriv                          !< return parallel derivative

        ! local variables
        complex(dp), allocatable :: xuq_arr(:,:,:,:)                    ! dummy variable to use array version

        ! allocate array version of XUQ
        allocate(xuq_arr(size(xuq,1),size(xuq,2),size(xuq,3),1))

        ! call array version
        ierr = calc_xuq_arr(mds_x,mds_sol,grid_eq,grid_x,grid_sol,eq_1,eq_2,x,&
            &sol,x_id,xuq_style,[time],xuq_arr,deriv=deriv)
        chckerr('')

        ! copy array to individual XUQ
        xuq = xuq_arr(:,:,:,1)

        ! deallocate array version of XUQ
        deallocate(xuq_arr)
    end function calc_xuq_ind

    !> Calculate  the  total version  of  the  solution  vector from  the  local
    !! version.
    !!
    !! This is the  solution vector for all of the  possible mode numbers, which
    !! can be different from the local mode numbers for X style 2 (fast):
    !!  -  local: the  size of  the saved  perturbation and  solution variables,
    !!  prescribed  by the  user as  input:
    !!      - either directly through \c n_mod_X
    !!      -  or through  the limits  on the  modes using  \c min_sec_X  and \c
    !!      max_sec_X.
    !!  - total: all  the possible modes that can resonate  in the plasma, which
    !!  can be  different from the  local number for \c  X_style 2, as  the mode
    !!  numbers depend on the normal coordinate.
    !!
    !! If the output variable is not allocated, it is done here.
    !!
    !! Optionally  a  derivative  can  be requested,  depending  on  the  normal
    !! discretization order.
    !!
    !! \note Need to provide the grid in which mds is tabulated.
    !!
    !! \return ierr
    integer function calc_tot_sol_vec(mds,grid_sol,sol_vec_loc,sol_vec_tot,&
        &deriv) result(ierr)
        use num_vars, only: x_style, norm_disc_prec_sol
        use x_vars, only: n_mod_x
        use num_utilities, only: spline

        character(*), parameter :: rout_name = 'calc_tot_sol_vec'

        ! input / output
        type(modes_type), intent(in) :: mds                             !< general modes variables
        type(grid_type), intent(in) :: grid_sol                         !< solution grid
        complex(dp), intent(in) :: sol_vec_loc(:,:)                     !< solution vector for local resonating modes
```

```
        complex(dp), intent(inout), allocatable :: sol_vec_tot(:,:)             !< solution vector for all possible resonating modes
        integer, intent(in), optional :: deriv                                  !< order of derivative

        ! local variables
        character(len=max_str_ln) :: err_msg                                    ! error message
        integer :: n_mod_loc                                                    ! local number of modes that are used in the calculations
        integer :: n_mod_tot                                                    ! total number of modes
        integer :: ld                                                           ! counter
        integer :: kdl(2)                                                       ! kd limits
        integer :: ld_loc                                                       ! local ld
        integer :: n_r                                                          ! normal size of variables
        integer :: deriv_loc                                                    ! local derivative
        integer :: max_deriv                                                    ! maximum derivative
        integer :: min_range                                                    ! minimum range for derivative
        complex(dp), allocatable :: dsol_vec_loc(:)                             ! local Dsol_vec_tot in solution grid

        ! initialize ierr
        ierr = 0

        ! set n_r
        n_r = size(sol_vec_loc,2)

        ! set local derivative
        deriv_loc = 0
        if (present(deriv)) deriv_loc = deriv

        ! test
        select case (norm_disc_prec_sol)
            case (1)                                                            ! linear
                max_deriv = 1
                min_range = 2
            case (2)                                                            ! Akita hermite
                max_deriv = 1
                min_range = 4
            case (3)                                                            ! spline
                max_deriv = 3
                min_range = 4
            case default
                ierr = 1
                err_msg = 'normal discretization precision for solution &
                    &cannot be higher than 3 or lower than 0'
                chckerr(err_msg)
        end select
        if (deriv_loc.lt.0 .or. deriv_loc.gt.max_deriv) then
            ierr = 1
            err_msg = 'deriv must be between 0 and '//trim(i2str(max_deriv))//&
                &' for normal discretization '//trim(i2str(norm_disc_prec_sol))
            chckerr(err_msg)
        endif

        ! operations depending on X style
        select case (x_style)
            case (1)                                                            ! prescribed
                ! test allocation
                if (allocated(sol_vec_tot)) then
                    if (size(sol_vec_tot,1).ne.size(sol_vec_loc,1) .or. &
                        &size(sol_vec_tot,2).ne.size(sol_vec_loc,2)) then
                        ierr = 1
                        err_msg = 'Local and total solution vectors need to &
                            &be of the same length'
                        chckerr(err_msg)
                    end if
                else
                    allocate(sol_vec_tot(size(sol_vec_loc,1),n_r))
                end if

                ! copy or derivate
                if (deriv_loc.eq.0) then                                        ! copy
                    sol_vec_tot = sol_vec_loc
                else                                                            ! derivate
                    do ld = 1,size(sol_vec_tot,1)
                        ierr = spline(grid_sol%r_F,sol_vec_loc(ld,:),&
                            &grid_sol%r_F,sol_vec_tot(ld,:),&
                            &ord=norm_disc_prec_sol,deriv=deriv_loc)
                        chckerr('')
                    end do
                end if
            case (2)                                                            ! fast
                ! set local and total nr. of modes
                n_mod_loc = n_mod_x
                n_mod_tot = size(mds%sec,1)

                ! test allocation
                if (allocated(sol_vec_tot)) then
                    if (size(sol_vec_tot,1).ne.n_mod_tot .or. &
```

```
                        &size(sol_vec_tot,2).ne.size(sol_vec_loc,2)) then
                        ierr = 1
                        err_msg = 'Total solution vectors needs to be of &
                            &correct length '//trim(i2str(n_mod_tot))
                        chckerr(err_msg)
                    end if
                else
                    allocate(sol_vec_tot(n_mod_tot,n_r))
                end if

                ! track all possible modes and copy or derivate
                sol_vec_tot = 0._dp
                do ld = 1,size(mds%sec,1)
                    ld_loc = mds%sec(ld,1)-minval(mds%sec(:,1),1)+1
                    kdl = [mds%sec(ld,2),mds%sec(ld,3)]-grid_sol%i_min+1        ! normal range in solution vector
                    kdl = max(1,min(kdl,n_r))                                   ! limit to sol_vec_loc range
                    if (deriv_loc.eq.0) then
                        sol_vec_tot(ld_loc,kdl(1):kdl(2)) = &
                            &sol_vec_loc(mds%sec(ld,4),kdl(1):kdl(2))
                    else
                        allocate(dsol_vec_loc(kdl(2)-kdl(1)+1))
                        dsol_vec_loc = 0._dp

                        if (kdl(2)-kdl(1)+1.ge.min_range) then                  ! only calculate if enough points
                            ierr = spline(grid_sol%r_F(kdl(1):kdl(2)),&
                                &sol_vec_loc(mds%sec(ld,4),kdl(1):kdl(2)),&
                                &grid_sol%r_F(kdl(1):kdl(2)),dsol_vec_loc,&
                                &ord=norm_disc_prec_sol,deriv=deriv_loc)
                            chckerr('')
                        end if
                        sol_vec_tot(ld_loc,kdl(1):kdl(2)) = dsol_vec_loc

                        deallocate(dsol_vec_loc)
                    end if
                end do
            end select
        end function calc_tot_sol_vec

        !> Calculate  the  local version  of  the  solution  vector from  the  total
        !! version.
        !!
        !! \see See calc_tot_sol_vec().
        !!
        !! \return ierr
        integer function calc_loc_sol_vec(mds,i_min,sol_vec_tot,sol_vec_loc) &
            &result(ierr)
            use num_vars, only: x_style
            use x_vars, only: n_mod_x

            character(*), parameter :: rout_name = 'calc_loc_sol_vec'

            ! input / output
            type(modes_type), intent(in) :: mds                                 !< general modes variables
            integer, intent(in) :: i_min                                        !< \c i_min of grid in which variables are tabulated
            complex(dp), intent(in) :: sol_vec_tot(:,:)                         !< solution vector for all possible resonating modes
            complex(dp), intent(inout), allocatable :: sol_vec_loc(:,:)         !< solution vector for local resonating modes

            ! local variables
            character(len=max_str_ln) :: err_msg                                ! error message
            integer :: n_mod_loc                                                ! local number of modes that are used in the calculations
            integer :: n_mod_tot                                                ! total number of modes
            integer :: ld                                                       ! counter
            integer :: kdl(2)                                                   ! kd limits
            integer :: ld_loc                                                   ! local ld
            integer :: n_r                                                      ! normal size of variables

            ! set n_r
            n_r = size(sol_vec_tot,2)

            ! initialize ierr
            ierr = 0

            ! operations depending on X style
            select case (x_style)
                case (1)                                                        ! prescribed
                    ! test allocation
                    if (allocated(sol_vec_loc)) then
                        if (size(sol_vec_loc,1).ne.size(sol_vec_tot,1) .or. &
                            &size(sol_vec_loc,2).ne.size(sol_vec_tot,2)) then
                            ierr = 1
                            err_msg = 'Local and total solution vectors need to &
                                &be of the same length'
                            chckerr(err_msg)
```

```
                    end if
                else
                    allocate(sol_vec_loc(size(sol_vec_tot,1),n_r))
                end if
                sol_vec_loc = sol_vec_tot
            case (2)                                                    ! fast
                ! set local and total nr. of modes
                n_mod_loc = n_mod_x
                n_mod_tot = size(mds%sec,1)

                ! test allocation
                if (allocated(sol_vec_loc)) then
                    if (size(sol_vec_loc,1).ne.n_mod_loc .or. &
                        &size(sol_vec_loc,2).ne.size(sol_vec_tot,2)) then
                        ierr = 1
                        err_msg = 'Total solution vectors needs to be of &
                            &correct length '//trim(i2str(n_mod_loc))
                        chckerr(err_msg)
                    end if
                else
                    allocate(sol_vec_loc(n_mod_loc,n_r))
                end if

                ! track all possible modes and copy
                sol_vec_loc = 0._dp
                do ld = 1,size(mds%sec,1)
                    ld_loc = mds%sec(ld,1)-minval(mds%sec(:,1),1)+1
                    kdl = [mds%sec(ld,2),mds%sec(ld,3)]-i_min+1         ! range in sol_vec_loc
                    kdl = max(1,min(kdl,n_r))                           ! limit to sol_vec_loc range
                    sol_vec_loc(mds%sec(ld,4),kdl(1):kdl(2)) = &
                        &sol_vec_tot(ld_loc,kdl(1):kdl(2))
                end do
        end select
    end function calc_loc_sol_vec

    !> Interpolation for a quantity V using splines.
    !!
    !! Optionally, a variable 'ivs_stat' is  returned that indicates whether the
    !! interpolation  was a  plain copy  (1), a  linear interpolation  (2) or  a
    !! spline interpolation (3).
    integer function interp_v_spline(V_i,V_o,r_i,r_o,extrap,ivs_stat) &
        &result(ierr)

        use num_utilities, only: spline

        character(*), parameter :: rout_name = 'interp_V_spline'

        ! input / output
        complex(dp), intent(in) :: v_i(:,:,:)                           ! input variable
        complex(dp), intent(out) :: v_o(:,:,:)                          ! output variable
        real(dp), intent(in) :: r_i(:)                                  ! input r
        real(dp), intent(in) :: r_o(:)                                  ! output r
        logical, intent(in) :: extrap                                   ! extrapolation possible
        integer, intent(out), optional :: ivs_stat                      ! which method was used (1: copy, 2: linear, 3: spline)

        ! local variables
        integer :: id, jd, kd                                           ! counters
        character(len=max_str_ln) :: err_msg                            ! error message

        ! initialize ierr
        ierr = 0

#if ldebug
        ! test sizes
        if (size(r_i).ne.size(v_i,3)) then
            ierr = 1
            err_msg = 'r_i and V_i are not compatible'
            chckerr(err_msg)
        end if
        if (size(r_o).ne.size(v_o,3)) then
            ierr = 1
            err_msg = 'r_o and V_o are not compatible'
            chckerr(err_msg)
        end if
        if (size(v_i,1).ne.size(v_o,1) .or. size(v_i,2).ne.size(v_o,2)) then
            ierr = 1
            err_msg = 'V_i and V_o are not compatible'
            chckerr(err_msg)
        end if
#endif

        ! interpolate depending on normal size
        select case (size(r_i))
            case (1)                                                    ! copy directly
                do kd = 1,size(v_o,3)
                    v_o(:,:,kd) = v_i(:,:,1)
```

```
                end do
                if (present(ivs_stat)) ivs_stat = 1
            case (2)                                                    ! manual linear (EZspline needs >= 3 pts)
                do kd = 1,size(v_o,3)
                    v_o(:,:,kd) = v_i(:,:,1) + &
                        &(v_i(:,:,2) - v_i(:,:,1)) / (r_i(2) - r_i(1)) * &
                        &(r_o(kd) - r_i(1))
                end do
                if (present(ivs_stat)) ivs_stat = 2
            case (3)                                                    ! linear via EZspline
                do jd = 1,size(v_i,2)
                    do id = 1,size(v_i,1)
                        ierr = spline(r_i,v_i(id,jd,:),r_o,v_o(id,jd,:),&
                            &ord=1,extrap=extrap)
                        chckerr('')
                    end do
                end do
                if (present(ivs_stat)) ivs_stat = 2
            case (4:)                                                   ! spline
                do jd = 1,size(v_i,2)
                    do id = 1,size(v_i,1)
                        ierr = spline(r_i,v_i(id,jd,:),r_o,v_o(id,jd,:),&
                            &ord=3,extrap=extrap)
                        chckerr('')
                    end do
                end do
                if (present(ivs_stat)) ivs_stat = 3
#if ldebug
            case default
                ierr = 1
                chckerr('Need more than 0 points')
#endif
        end select

#if ldebug
        ! visualize
        do jd = 1,size(v_i,2)
            do id = 1,size(v_i,1)
                if (debug_interp_v_spline) then
                    call plot_comp('real part',r_i,rp(v_i(id,jd,:)),&
                        &r_o,rp(v_o(id,jd,:)))
                    call plot_comp('imaginary part',r_i,ip(v_i(id,jd,:)),&
                        &r_o,ip(v_o(id,jd,:)))
                end if
            end do
        end do
    contains
        !> \private Plot to compare interpolation to original variables
        subroutine plot_comp(plot_name,x_i,y_i,x_o,y_o)
            ! input / output
            character(len=*), intent(in) :: plot_name                   ! name of plot
            real(dp), intent(in) :: x_i(:)                              ! x input
            real(dp), intent(in) :: y_i(:)                              ! y input
            real(dp), intent(in) :: x_o(:)                              ! x output
            real(dp), intent(in) :: y_o(:)                              ! y output

            ! local variables
            real(dp), allocatable :: x_plot(:,:)                        ! x for plot
            real(dp), allocatable :: y_plot(:,:)                        ! y for plot
            character(len=max_str_ln) :: var_names(2)                   ! variable names

            allocate(x_plot(max(size(x_i),size(x_o)),2))
            allocate(y_plot(max(size(x_i),size(x_o)),2))

            x_plot(1:size(x_i),1) = x_i
            x_plot(size(x_i):size(x_plot,1),1) = x_i(size(x_i))
            x_plot(1:size(x_o),2) = x_o
            x_plot(size(x_o):size(x_plot,1),2) = x_o(size(x_o))

            y_plot(1:size(x_i),1) = y_i(:)
            y_plot(size(x_i):size(y_plot,1),1) = y_i(size(x_i))
            y_plot(1:size(x_o),2) = y_o(:)
            y_plot(size(x_o):size(y_plot,1),2) = y_o(size(x_o))

            var_names(1) = 'input '//plot_name
            var_names(2) = 'output '//plot_name
            call print_ex_2d(var_names,'',y_plot,x=x_plot)
        end subroutine plot_comp
#endif
    end function interp_v_spline
end module sol_utilities
```

D.80 /Users/toonweyens/Code/PB3D/Modules/sol_vars.f90 File Reference

Interfaces and Types

- type sol_vars::sol_type
  solution type

Modules

- module sol_vars
  Variables pertaining to the solution quantities.

Functions/Subroutines

- subroutine sol_vars::init_sol (sol, mds, grid_sol, n_ev, lim_sec_x)
  Initialize a solution type and allocate the variables.
- subroutine sol_vars::dealloc_sol (sol)
  Deallocates solution variables.

Variables

- integer, public sol_vars::n_alloc_sols
  nr. of allocated grids

D.81 sol_vars.f90

Go to the documentation of this file.

```
!------------------------------------------------------------------------!
!> Variables pertaining to the solution quantities.
!------------------------------------------------------------------------!
module sol_vars
#include <PB3D_macros.h>
    use str_utilities
    use messages
    use num_vars, only: dp, max_str_ln, iu, weight_dp
    use grid_vars, only: grid_type
    use x_vars, only: modes_type
    use output_ops

    implicit none

    private
#if ldebug
    public n_alloc_sols
#endif

    ! global variables
#if ldebug
    integer :: n_alloc_sols                                          !< nr. of allocated grids \ldebug
#endif

    !> solution type
    !!
    !! The arrays here are of the form:
    !!  - \c val: <tt>(1:n_EV)</tt>
    !!  - \c vec: <tt>(1:n_mod,1:loc_n_r,1:n_EV)</tt>
```

```
    type, public :: sol_type
        integer :: n_mod                                                    !< size of n and m (nr. of modes)
        integer :: lim_sec_x(2)                                             !< limits of \c m_X (pol. flux) or \c n_X (tor. flux)
        integer, allocatable :: n(:,:)                                      !< vector of toroidal mode numbers
        integer, allocatable :: m(:,:)                                      !< vector of poloidal mode numbers
        complex(dp), allocatable :: vec(:,:,:)                              !< Eigenvector solution
        complex(dp), allocatable :: val(:)                                  !< Eigenvalue solution
#if ldebug
        real(dp) :: estim_mem_usage                                         !< estimated memory usage \ldebug
#endif
    contains
        !> initialize
        procedure :: init => init_sol
        !> deallocate
        procedure :: dealloc => dealloc_sol
    end type

contains
    !> \public Initialize a solution type and allocate the variables.
    !!
    !! The number of modes as well as \c n and \c m are also set up.
    !!
    !! Optionally, the secondary mode number can  be specified (\c m if poloidal
    !! flux is used as normal coordinate and  c n if toroidal flux). By default,
    !! they are taken from the global \c X_vars variables.
    !!
    !! \note If the lowest limits of  the grid is not 1 (e.g. <tt>grid_sol%i_min
    !! = 1</tt> for first process), the input variable \c i_min should be set to
    !! set correctly. For a full grid, it should be set to 1.
    subroutine init_sol(sol,mds,grid_sol,n_EV,lim_sec_X)
        use x_vars, only: set_nm_x
#if ldebug
        use num_vars, only: print_mem_usage, rank
#endif

        ! input / output
        class(sol_type), intent(inout) :: sol                               !< solution variables
        type(modes_type), intent(in) :: mds                                 !< general modes variables
        type(grid_type), intent(in) :: grid_sol                             !< solution grid
        integer, intent(in) :: n_EV                                         !< nr. of Eigenvalues
        integer, intent(in), optional :: lim_sec_X(2)                       !< limits of \c m_X (pol. flux) or \c n_X (tor. flux)

#if ldebug
        ! initialize memory usage
        if (print_mem_usage) sol%estim_mem_usage = 0._dp
#endif

        ! set mode numbers
        call set_nm_x(mds,grid_sol,sol%lim_sec_X,sol%n,sol%m,lim_sec_x)

        ! set n_mod
        sol%n_mod = size(sol%n,2)

        ! allocate val
        allocate(sol%val(1:n_ev))

        ! allocate vec
        allocate(sol%vec(1:sol%n_mod,1:grid_sol%loc_n_r,1:n_ev))

#if ldebug
        ! set estimated memory usage
        if (print_mem_usage) sol%estim_mem_usage = sol%estim_mem_usage + &
            &n_ev * (1 + sol%n_mod*grid_sol%loc_n_r)

        ! increment n_alloc_sols
        n_alloc_sols = n_alloc_sols + 1
        if (print_mem_usage) call writo('[rank '//trim(i2str(rank))//&
            &' - Expected memory usage of sol: '//&
            &trim(r2strt(sol%estim_mem_usage*weight_dp*2))//' kB]',alert=.true.)
#endif
    end subroutine init_sol

    !> \public Deallocates solution variables.
    !!
    !! \note intent(out) automatically deallocates the variable
    subroutine dealloc_sol(sol)
#if ldebug
        use num_vars, only: rank, print_mem_usage
#endif

        ! input / output
        class(sol_type), intent(inout) :: sol                               !< solution variables to be deallocated

#if ldebug
        ! local variables
```

```
        integer :: mem_diff                                                     ! difference in memory
        real(dp) :: estim_mem_usage                                             ! estimated memory usage

        ! memory usage before deallocation
        if (print_mem_usage) then
            mem_diff = get_mem_usage()
            estim_mem_usage = sol%estim_mem_usage
        end if
#endif

        ! deallocate allocatable variables
        call dealloc_sol_final(sol)

#if ldebug
        ! decrement n_alloc_sols
        n_alloc_sols = n_alloc_sols - 1

        ! memory usage difference after deallocation
        if (print_mem_usage) then
            mem_diff = mem_diff - get_mem_usage()
            call writo('[Rank '//trim(i2str(rank))//' - liberated '//&
                &trim(i2str(mem_diff))//'kB deallocating sol ('//&
                &trim(i2str(nint(100*mem_diff/&
                &(estim_mem_usage*weight_dp*2)))))//&
                &'% of estimated)]',alert=.true.)
        end if
#endif
    contains
        ! Note: intent(out) automatically deallocates the variable
        !> \private
        subroutine dealloc_sol_final(sol)
            ! input / output
            type(sol_type), intent(out) :: sol                                  ! solution to be deallocated
        end subroutine dealloc_sol_final
    end subroutine dealloc_sol
end module sol_vars
```

D.82 /Users/toonweyens/Code/PB3D/Modules/str_utilities.f90 File Reference

Modules

- module str_utilities

  Operations on strings.

Functions/Subroutines

- elemental character(len=max_str_ln) function, public str_utilities::i2str (k)

  Convert an integer to string.
- elemental character(len=max_str_ln) function, public str_utilities::ii2str (k)

  Convert an integer to string.
- elemental character(len=max_str_ln) function, public str_utilities::r2str (k)

  Convert a real (double) to string.
- elemental character(len=max_str_ln) function, public str_utilities::r2strt (k)

  Convert a real (double) to string.
- elemental character(len=max_str_ln) function, public str_utilities::c2str (k)

  Convert a complex (double) to string.
- elemental character(len=max_str_ln) function, public str_utilities::c2strt (k)

  Convert a complex (double) to string.
- character(len(input_string)) function, public str_utilities::strh2l (input_string)

  Convert a string to lowercase.
- character(len(input_string)) function, public str_utilities::strl2h (input_string)

  convert a string to uppercase.
- character((len(input_strings)+2) ∗size(input_strings)) function, public str_utilities::merge_strings (input_strings)

  Merge array of strings.

D.83 str_utilities.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Operations on strings.
!-----------------------------------------------------------------------------!
module str_utilities
#include <wrappers.h>
    use num_vars, only: dp, max_str_ln
    implicit none
    private
    public i2str, ii2str, r2str, r2strt, c2str, c2strt, strh2l, strl2h, &
        &merge_strings

contains
    !> Convert an integer to string.
    !!
    !! \see from
    !! <http://stackoverflow.com/questions/1262695/converting-integers-to-strings-in-fortran>
    elemental character(len=max_str_ln) function i2str(k)
        ! input / output
        integer, intent(in) :: k                                        !< integer to convert

        write (i2str, *) k
        i2str = adjustl(i2str)
    end function i2str
    !> Convert an integer to string.
    !!
    !! Version with kind 8 integers.
    !!
    !! \see See i2str().
    elemental character(len=max_str_ln) function ii2str(k)
        ! input / output
        integer(kind=8), intent(in) :: k                                !< integer to convert

        write (ii2str, *) k
        ii2str = adjustl(ii2str)
    end function ii2str

    !> Convert a real (double) to string
    !!
    !! \note See  <http://www.fortran90.org/src/best-practices.html> how  to not
    !! lose precision.
    elemental character(len=max_str_ln) function r2str(k)
        ! input / output
        real(dp), intent(in) :: k                                       !< real to convert

        write (r2str, '(ES23.16)') k
        r2str = adjustl(r2str)
    end function r2str
    !> Convert a real (double) to string
    !!
    !! Version with less precise output.
    !!
    !! \see See r2str().
    elemental character(len=max_str_ln) function r2strt(k)
        ! input / output
        real(dp), intent(in) :: k                                       !< real to convert

        write (r2strt, '(ES9.2)') k
        r2strt = adjustl(r2strt)
    end function r2strt

    !> Convert a complex (double) to string
    !!
    !! \note See  <http://www.fortran90.org/src/best-practices.html> how  to not
    !! lose precision.
    elemental character(len=max_str_ln) function c2str(k)
        ! input / output
        complex(dp), intent(in) :: k                                    !< complex to convert

        ! local variables
        character(len=max_str_ln) :: dum_str                            ! dummy string

        write (c2str, '(ES23.16)') rp(k)
        write (dum_str, '(ES23.16)') abs(imag(k))
        if (imag(k).lt.0) then
            c2str = trim(c2str)//' -'
        else
            c2str = trim(c2str)//' +'
        end if
        c2str = trim(c2str)//' '//dum_str
        c2str = adjustl(c2str)
    end function c2str
```

```fortran
    !> Convert a complex (double) to string
    !!
    !! Version with less precise output.
    !!
    !! \see See c2str().
    elemental character(len=max_str_ln) function c2strt(k)
        ! input / output
        complex(dp), intent(in) :: k                                        !< complex to convert

        ! local variables
        character(len=max_str_ln) :: dum_str                                ! dummy string

        write (c2strt, '(ES9.2)') rp(k)
        write (dum_str, '(ES9.2)') abs(imag(k))
        if (imag(k).lt.0) then
            c2strt = trim(c2strt)//' -'
        else
            c2strt = trim(c2strt)//' +'
        end if
        c2strt = trim(c2strt)//' '//trim(dum_str)//' i'
        c2strt = adjustl(c2strt)
    end function c2strt

    !> Convert a string to lowercase.
    !!
    !! \see from \cite RedwineF90, figure 3.5B, pg 80.
    function strh2l(input_string) result(output_string)
        ! input / output
        character(*), intent(in)     :: input_string                        !< input string
        character(len(input_string)) :: output_string                       !< lowercase version

        ! local variables
        integer :: i, n
        character(*), parameter :: lower_case = 'abcdefghijklmnopqrstuvwxyz'
        character(*), parameter :: upper_case = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

        ! copy input string
        output_string = input_string

        ! convert case character by character
        do i = 1, len(output_string)
          n = index(upper_case, output_string(i:i))
          if ( n /= 0 ) output_string(i:i) = lower_case(n:n)
        end do
    end function strh2l
    !> convert a string to uppercase.
    !!
    !! \see See strh2l()
    function strl2h(input_string) result(output_string)
        ! input / output
        character(*), intent(in)     :: input_string                        !< input string
        character(len(input_string)) :: output_string                       !< uppercase version

        ! local variables
        integer :: i, n
        character(*), parameter :: lower_case = 'abcdefghijklmnopqrstuvwxyz'
        character(*), parameter :: upper_case = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

        ! copy input string
        output_string = input_string

        ! convert case character by character
        do i = 1, len(output_string)
          n = index(lower_case, output_string(i:i))
          if ( n /= 0 ) output_string(i:i) = upper_case(n:n)
        end do
    end function strl2h

    !> Merge array of strings.
    function merge_strings(input_strings)
        ! input / output
        character(*), intent(in) :: input_strings(:)                        !< array of strings
        character((len(input_strings)+2)*size(input_strings)) :: merge_strings  !< merged string

        ! local variables
        integer :: id                                                       ! counter

        ! start with first string
        if (size(input_strings).gt.0) then
            merge_strings = trim(input_strings(1))

            ! loop over next strings
            do id = 2,size(input_strings)
                merge_strings = trim(merge_strings)//', '//&
                    &trim(input_strings(id))
            end do
        else
```

```
            merge_strings = ''
        end if
    end function merge_strings
end module str_utilities
```

D.84 /Users/toonweyens/Code/PB3D/Modules/test.f90 File Reference

Modules

- module test

  Generic tests.

Functions/Subroutines

- integer function, public test::generic_tests ()

  Performs generic tests.
- subroutine plot_y (x, x_int, y, y_int, deriv)

D.84.1 Function/Subroutine Documentation

D.84.1.1 plot_y()

```
subroutine test_splines::plot_y (
                real(dp), dimension(:), intent(in) x,
                real(dp), dimension(:), intent(in) x_int,
                real(dp), dimension(:), intent(in) y,
                real(dp), dimension(:), intent(in) y_int,
                integer, intent(in) deriv)
```

Definition at line 308 of file test.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

D.85 test.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------!
!> Generic tests.
!-----------------------------------------------------------------------!
module test
#include <PB3D_macros.h>
```

```
#include <wrappers.h>
    use str_utilities
    use output_ops
    use messages
    use num_vars, only: pi, dp, max_str_ln, iu
    use input_utilities

    implicit none
    private
#if ldebug
    public generic_tests
#endif

#if ldebug
contains
    !> Performs generic tests.
    !!
    !! These tests  are run in interactive  fashion by running the  program with
    !! the <tt>--test</tt> command-line option.
    !! \see init_files()
    !!
    !! \note
    !!  -#    It   is    probably   also    recommended   to    run   it    with
    !!  <tt>--do_execute_command_line</tt>  in order  to generate  the plots  in
    !!  real time.
    !!  -#  It has  not  been  tested whether  testing  routines "pollutes"  the
    !!  further simulations.  It is  therefore best  to not  use the  tests when
    !!  doing a real run.
    !!
    !! \ldebug
    !!
    !! \return ierr
    integer function generic_tests() result(ierr)
        use num_vars, only: prog_style

        character(*), parameter :: rout_name = 'generic_tests'

        ! initialize ierr
        ierr = 0

        call writo('Test smoothed second derivative?')
        if (get_log(.false.)) then
            ierr = test_calc_d2_smooth()
            chckerr('')
            call pause_prog
        end if

        call writo('Test splines?')
        if (get_log(.false.)) then
            ierr = test_splines()
            chckerr('')
            call pause_prog
        end if

        call writo('Test lock system?')
        if (get_log(.false.)) then
            ierr = test_lock()
            chckerr('')
            call pause_prog
        end if

        call writo('Test calculation of RZL?')
        if (get_log(.false.)) then
            ierr = test_calc_rzl()
            chckerr('')
            call pause_prog
        end if

        call writo('Test calculation of toroidal functions?')
        if (get_log(.false.)) then
            ierr = test_tor_fun()
            chckerr('')
            call pause_prog
        end if

        ! select according to program style
        select case (prog_style)
            case(1)                                                         ! PB3D
                ! do nothing
            case(2)                                                         ! POST
                call writo('Test reading of subsets?')
                if (get_log(.false.)) then
                    ierr = test_read_hdf5_subset()
                    chckerr('')
                    call pause_prog
                end if

```

```
                call writo('Test calculation of volume integral?')
                if (get_log(.false.)) then
                    ierr = test_calc_int_vol()
                    chckerr('')
                    call pause_prog
                end if
        end select
    end function generic_tests

    !> Test calc_d2_smooth().
    !!
    !! \return ierr
    integer function test_calc_d2_smooth() result(ierr)
        use num_utilities, only: calc_d2_smooth

        character(*), parameter :: rout_name = 'test_calc_D2_smooth'

        ! local variables
        integer :: kd
        integer :: nx
        integer :: fil_n
        integer :: style
        integer :: sym_m
        real(dp), allocatable :: x(:)
        real(dp), allocatable :: y(:,:)
        real(dp), allocatable :: d2y(:)
        logical :: ready

        ! initialize ierr
        ierr = 0

        ! user input
        call writo('testing Holoborodko''s smoothed second derivatives')
        call lvl_ud(1)

        ! set up function
        call writo('how many points?')
        nx = get_int(lim_lo=20)
        allocate(x(nx))
        allocate(y(nx,0:2))
        allocate(d2y(nx))
        do kd = 1,nx
            x(kd) = (kd-1._dp)/(nx-1._dp)                                    ! 0..1
            y(kd,0) = sin(2*pi*x(kd))
            y(kd,1) = 2*pi*cos(2*pi*x(kd))
            y(kd,2) = -(2*pi)**2*sin(2*pi*x(kd))
            !!y(kd,0) = x(kd)**3
            !!y(kd,1) = 3._dp*x(kd)**2
            !!y(kd,2) = 6._dp*x(kd)
        end do

        ready = .false.
        do while (.not.ready)
            ! get filter length
            call writo('filter length?')
            fil_n = get_int(lim_lo=5,lim_hi=nx)
            sym_m = (fil_n-1)/2

            ! get style
            call writo('style?')
            call lvl_ud(1)
            call writo('1: central')
            call writo('2: left')
            call lvl_ud(-1)
            style = get_int(lim_lo=1,lim_hi=2)

            ! calculate derivative
            ierr = calc_d2_smooth(fil_n,x,y(:,0),d2y,style)
            chckerr('')

            ! plot
            call print_ex_2d(['y  ','D2y'],'',reshape([y(:,2),d2y],[nx,2]),&
                &x=reshape(x,[nx,1]))
            select case (style)
                case (1)
                    call print_ex_2d('y-D2y','',&
                        &y(sym_m+1:nx-sym_m,2)-d2y(sym_m+1:nx-sym_m),&
                        &x=x(sym_m+1:nx-sym_m))
                case (2)
                    call print_ex_2d('y-D2y','',&
                        &y(2*sym_m+1:nx,2)-d2y(2*sym_m+1:nx),&
                        &x=x(2*sym_m+1:nx))
            end select

            call writo('repeat?')
            ready = .not.get_log(.true.)
        end do
```

```fortran

        call lvl_ud(-1)
    end function test_calc_d2_smooth

    !> Test spline().
    !!
    !! \return ierr
    integer function test_splines() result(ierr)
        use num_utilities, only: spline
        use num_vars, only: tol_zero

        character(*), parameter :: rout_name = 'test_splines'

        ! local variables
        integer :: kd, id
        integer :: nx
        integer :: nx_int
        integer :: ord
        integer :: max_deriv
        integer :: bcs(2)                                                   ! boundary conditions
        real(dp) :: bcs_val(2)                                              ! value of derivative at boundary conditions
        real(dp) :: extrap_fraction
        real(dp), allocatable :: x(:)
        real(dp), allocatable :: x_int(:)
        real(dp), allocatable :: y(:,:)
        real(dp), allocatable :: y_int(:)
        logical :: ready
        logical :: extrap
        character(len=5) :: side_str(2)

        ! initialize ierr
        ierr = 0

        ! user input
        call writo('testing splines')
        call lvl_ud(1)

        ! set up function
        nx = 10
        allocate(x(nx))
        allocate(y(nx,0:3))
        do kd = 1,nx
            x(kd) = (kd-1._dp)/(nx-1._dp)                                   ! 0..1
            y(kd,0) = sin(2*pi*x(kd))
            y(kd,1) = 2*pi*cos(2*pi*x(kd))
            y(kd,2) = -(2*pi)**2*sin(2*pi*x(kd))
            y(kd,3) = -(2*pi)**3*cos(2*pi*x(kd))
        end do

        ! set up interpolated function
        nx_int = 400
        allocate(x_int(nx_int))
        allocate(y_int(nx_int))

        ready = .false.
        do while (.not.ready)
            ! get order
            call writo('order?')
            call lvl_ud(1)
            call writo('1: linear')
            call writo('2: akima hermite')
            call writo('3: cubic')
            call lvl_ud(-1)
            ord = get_int(lim_lo=1,lim_hi=3)

            ! get extrapolation fraction
            call writo('test extrapolation?')
            extrap = get_log(.false.)
            if (extrap) then
                call writo('Extrapolation fraction?')
                extrap_fraction = get_real(lim_lo=0._dp,lim_hi=10._dp)
            else
                extrap_fraction = -tol_zero
            end if

            side_str(1) = 'left'
            side_str(2) = 'right'
            if (ord.gt.1) then
                do kd = 1,2
                    call writo('boundary condition '//trim(side_str(kd))//&
                        &' = ?')
                    call lvl_ud(1)
                    call writo('-1: periodic')
                    call writo(' 0: not-a-knot')
                    call writo(' 1: first derivative prescribed')
                    call writo(' 2: second derivative prescribed')
                    call lvl_ud(-1)
```

```
                    bcs(kd) = get_int(lim_lo=-1,lim_hi=2)
                    if (bcs(kd).eq.1 .or. bcs(kd).eq.2) then
                        call writo('value of derivative of order '//&
                            &trim(i2str(bcs(kd))))
                        bcs_val(kd) = get_real()
                    else
                        bcs_val(kd) = 0._dp
                    end if
                end do
            else
                bcs = 0
                bcs_val = 0._dp
            end if

            select case (ord)
                case (1:2)
                    max_deriv = 1
                case (3)
                    max_deriv = 3
            end select

            ! set up x_int
            do kd = 1,nx_int
                x_int(kd) = -extrap_fraction + &
                    &(1+2*extrap_fraction)*(kd-1._dp)/(nx_int-1._dp)            ! 0..1 with possible extrapolation fraction
            end do

            ! calculate spline and possibly plot
            do id = 0,max_deriv
                ierr = spline(x,y(:,0),x_int,y_int,ord=ord,deriv=id,bcs=bcs,&
                    &bcs_val=bcs_val,extrap=extrap)                             ! use y(0)
                chckerr('')
                call plot_y(x,x_int,y(:,id),y_int,id)                           ! use y(id)
            end do

            call writo('repeat?')
            ready = .not.get_log(.true.)
        end do

        call lvl_ud(-1)
    contains
        subroutine plot_y(x,x_int,y,y_int,deriv)
            ! input / output
            real(dp), intent(in) :: x(:)
            real(dp), intent(in) :: x_int(:)
            real(dp), intent(in) :: y(:)
            real(dp), intent(in) :: y_int(:)
            integer, intent(in) :: deriv

            ! local variables
            integer :: nx, nx_int
            real(dp) :: minmax_y(2)
            real(dp), allocatable :: x_plot(:,:)
            real(dp), allocatable :: y_plot(:,:)
            character(len=max_str_ln), allocatable :: plot_names(:)

            ! set up plot variables
            nx = size(x)
            nx_int = size(x_int)
            allocate(x_plot(max(nx,nx_int),2))                                  ! (x_int,x)
            allocate(y_plot(max(nx,nx_int),2))                                  ! (y_int,y)

            ! x,y
            x_plot(1:nx_int,1) = x_int
            x_plot(nx_int+1:max(nx,nx_int),1) = x_int(nx_int)
            y_plot(1:nx_int,1) = y_int
            y_plot(nx_int+1:max(nx,nx_int),1) = y_int(nx_int)

            ! int x,y
            x_plot(1:nx,2) = x
            x_plot(nx+1:max(nx,nx_int),2) = x(nx)
            y_plot(1:nx,2) = y
            y_plot(nx+1:max(nx,nx_int),2) = y(nx)

            ! knots
            minmax_y(1) = minval(y_plot(:,1:2))
            minmax_y(2) = maxval(y_plot(:,1:2))

            ! plot
            allocate(plot_names(2))
            plot_names(1) = 'D'//trim(i2str(deriv))//'y_int'
            plot_names(2) = 'D'//trim(i2str(deriv))//'y'
            call print_ex_2d(plot_names,plot_names(1),y_plot,x=x_plot)
        end subroutine plot_y
    end function test_splines

    !> Test lock system.
```

```
!!
!! \return ierr
integer function test_lock() result(ierr)
    use mpi_vars, only: lock_type
    use mpi_utilities, only: lock_req_acc, lock_return_acc, wait_mpi, &
        &lock_header, lock_wl_change, &
        &debug_lock
    use num_vars, only: rank

    character(*), parameter :: rout_name = 'test_lock'

    ! local variables
    type(lock_type) :: lock
    integer :: wait_time
    integer(kind=8) :: sleep_time(2)
    integer :: testcase
    integer :: n_cycles
    integer :: id
    logical :: blocking
    logical :: red_lat
    integer, allocatable :: wl(:)
    character(len=max_str_ln) :: title

    ! initialize ierr
    ierr = 0

    ! debug messages
    debug_lock = .true.

    call writo('which test?')
    call lvl_ud(1)
    call writo('1. blocking')
    call writo('2. non-blocking')
    call writo('3. blocking and non-blocking')
    testcase = get_int(lim_lo=1,lim_hi=3)
    call lvl_ud(-1)

    call writo('How many cycles?')
    n_cycles = get_int(lim_lo=1)

    if (testcase.eq.2 .or. testcase.eq.3) then
        call writo('Do you want to check whether locking and unlocking the &
            &window multiple times by the master helps reduce latency?')
        red_lat = get_log(.false.)
    end if

    ! set name
    select case (testcase)
        case (1)
            title = 'blocking'
        case (2)
            title = 'non-blocking'
        case (3)
            title = 'blocking and non-blocking'
    end select

    ! create lock
    ierr = lock%init(100)
    chckerr('')

    ! test blocking access
    call writo('Testing '//trim(title)//' access')
    call lvl_ud(1)
    call writo('Every process immediately requests access and then &
        &waits some seconds between 0 and 3')

    ! set name
    select case (testcase)
        case (1)
            call writo('Every process is blocking')
        case (2)
            call writo('Every process is non-blocking')
        case (3)
            call writo('In the first cycle, every process is non-blocking')
            call writo('Afterwards, each process becomes blocking &
                &sequentially')
    end select

    ! synchronize MPI
    ierr = wait_mpi()
    chckerr('')

    do id = 1,n_cycles
        select case (testcase)
            case (1)                                                    ! blocking
                blocking = .true.
            case (2)                                                    ! non-blocking
```

```
                    blocking = .false.
                case (3)                                                    ! both
                    if (id.eq.rank+2) then
                        blocking = .true.
                    else
                        blocking = .false.
                    end if
            end select
            ierr = lock_req_acc(lock,blocking)
            chckerr('')

            wait_time = random_int([0,3])
            write(*,*) trim(lock_header(lock)),'sleeps ',&
                &wait_time,' seconds'
            if (red_lat .and. rank.eq.0 .and. wait_time.gt.0) then
                call system_clock(sleep_time(1))
                call system_clock(sleep_time(2))
                do while (1.e-9_dp*(sleep_time(2)-sleep_time(1)).lt.wait_time)
                    debug_lock = .false.
                    ierr = lock_wl_change(2,lock%blocking,lock,wl)
                    chckerr('')
                    call system_clock(sleep_time(2))
                end do
                debug_lock = .true.
            else
                call sleep(wait_time)
            endif
            write(*,*) trim(lock_header(lock)),'has slept ',&
                &wait_time,' seconds, returning lock'

            ierr = lock_return_acc(lock)
            chckerr('')
        end do

        ! clean up
        ierr = lock%dealloc()
        chckerr('')

        ! synchronize MPI
        ierr = wait_mpi()
        chckerr('')

        call lvl_ud(-1)
    contains
        ! not very high quality random number between the limits
        !> \private
        integer function random_int(lim) result(res)
            ! input / output
            integer, intent(in) :: lim(2)

            ! local variables
            integer :: seed(33)
            real(dp) :: r_nr
            integer :: n_steps = 5
            integer :: id

            do id = 1,n_steps
                ! random number for seed
                call random_number(r_nr)

                ! set seed
                seed = int(1000_dp*r_nr*(rank+1))
                call random_seed(put=seed)
            end do

            ! random numbeer
            call random_number(r_nr)
            res = lim(1) + floor((lim(2)-lim(1)+1)*r_nr)
        end function random_int
    end function test_lock

    !> Tests the calculation of R, Z, and L.
    !!
    !! \return ierr
    integer function test_calc_rzl() result(ierr)
        use pb3d_ops, only: reconstruct_pb3d_in
        use num_vars, only: eq_style

        character(*), parameter :: rout_name = 'test_calc_RZL'

        ! initialize ierr
        ierr = 0

        ! preliminary things
        ierr = reconstruct_pb3d_in('in')                                    ! reconstruct miscellaneous PB3D output
    variables
        chckerr('')
```

```fortran

        ! select according to equilibrium style
        select case (eq_style)
            case(1)                                                         ! VMEC
                ierr = test_calc_rzl_vmec()
                chckerr('')
            case(2)                                                         ! HELENA
                ! do nothing
        end select

        ! clean up
        call dealloc_in()
    contains
        !> \private
        integer function test_calc_rzl_vmec() result(ierr)
            use num_vars, only: rank
            use grid_vars, only: n_r_eq, grid_type
            use grid_utilities, only: calc_eqd_grid
            use vmec_utilities, only: calc_trigon_factors, fourier2real
            use vmec_vars, only: r_v_c, r_v_s, z_v_c, z_v_s, l_v_c, l_v_s, &
                &is_asym_v

            character(*), parameter :: rout_name = 'test_calc_RZL_VMEC'

            ! local variables
            integer :: id, jd                                               ! counters
            integer :: dims(3)                                              ! dimensions of grid
            integer :: norm_id                                              ! normal point for which to do the analysis
            real(dp) :: grid_lims(2,2)                                      ! limits on grid
            real(dp), allocatable :: norm(:,:,:)                            ! copy of normal grid variable
            real(dp), allocatable :: R(:,:,:), Z(:,:,:), L(:,:,:)           ! local R, Z and L
            type(grid_type) :: grid                                         ! grid
            integer :: deriv(3)                                             ! local derivative
            integer :: max_deriv(3)                                         ! maximum derivative

            ! user output
            call writo('Going to test the calculation of R, Z and L')
            call lvl_ud(1)

            ! initialize ierr
            ierr = 0

            if (rank.eq.0) then
                ! set normal point
                call writo('For which normal point do you want the analysis?')
                norm_id = get_int(lim_lo=1,lim_hi=n_r_eq)

                ! set dimensions [theta,zeta,r]
                dims = [1,1,1]
                do id = 2,3
                    call writo('grid size in dimension '//trim(i2str(id))//'?')
                    dims(id-1) = get_int(lim_lo=1)

                    call writo('limits for angle '//trim(i2str(id))//' [pi]?')
                    grid_lims(1,id-1) = get_real()*pi
                    grid_lims(2,id-1) = get_real(lim_lo=grid_lims(1,id-1)/pi)*pi
                end do

                ierr = grid%init(dims)
                chckerr('')

                ! create grid
                allocate(norm(dims(1),dims(2),dims(3)))
                ierr = calc_eqd_grid(grid%theta_E,grid_lims(1,1),&
                    &grid_lims(2,1),1)
                chckerr('')
                ierr = calc_eqd_grid(grid%zeta_E,grid_lims(1,2),&
                    &grid_lims(2,2),2)
                chckerr('')
                norm = (norm_id-1._dp)/(n_r_eq-1)
                grid%r_E = norm(1,1,:)

                ! set maximum derivatives [r,theta,zeta]
                max_deriv(1) = 0
                do id = 2,3
                    call writo('maximum derivative in dimension '//&
                        &trim(i2str(id))//'?')
                    max_deriv(id) = get_int(lim_lo=0,lim_hi=3)
                end do

                ! calculate R, Z and L
                allocate(r(dims(1),dims(2),dims(3)))
                allocate(z(dims(1),dims(2),dims(3)))
                allocate(l(dims(1),dims(2),dims(3)))
                ierr = calc_trigon_factors(grid%theta_E,grid%zeta_E,&
                    &grid%trigon_factors)
                chckerr('')
```

```
                do jd = 0,max_deriv(3)
                    do id = 0,max_deriv(2)
                        deriv = [0,id,jd]
                        call writo('Calculating for deriv = ['//&
                            &trim(i2str(deriv(1)))//','//&
                            &trim(i2str(deriv(2)))//','//&
                            &trim(i2str(deriv(3)))//']')
                        call lvl_ud(1)
                        ierr = fourier2real(r_v_c(:,norm_id:norm_id,deriv(1)),&
                            &r_v_s(:,norm_id:norm_id,deriv(1)),&
                            !&grid%theta_E,grid%zeta_E,&
                            &grid%trigon_factors,&
                            &r,sym=[.true.,is_asym_v],&
                            &deriv=[deriv(2),deriv(3)])
                        chckerr('')
                        ierr = fourier2real(z_v_c(:,norm_id:norm_id,deriv(1)),&
                            &z_v_s(:,norm_id:norm_id,deriv(1)),&
                            !&grid%theta_E,grid%zeta_E,&
                            &grid%trigon_factors,&
                            &z,sym=[is_asym_v,.true.],&
                            &deriv=[deriv(2),deriv(3)])
                        chckerr('')
                        ierr = fourier2real(l_v_c(:,norm_id:norm_id,deriv(1)),&
                            &l_v_s(:,norm_id:norm_id,deriv(1)),&
                            !&grid%theta_E,grid%zeta_E,&
                            &grid%trigon_factors,&
                            &l,sym=[is_asym_v,.true.],&
                            &deriv=[deriv(2),deriv(3)])
                        chckerr('')
                        call plot_hdf5(['R','Z','L'],'TEST_RZL_0_'//&
                            &trim(i2str(id))//'_'//trim(i2str(jd)),&
                            &reshape([r,z,l],[dims,3]),&
                            &x=reshape([grid%theta_E],[dims,1]),&
                            &y=reshape([grid%zeta_E],[dims,1]),&
                            &z=reshape([norm],[dims,1]))
                        call lvl_ud(-1)
                    end do
                end do
            end if

            ! user output
            call lvl_ud(-1)
            call writo('Test complete')
        end function test_calc_rzl_vmec
    end function test_calc_rzl

    !> Test calculation of toroidal functions.
    !!
    !! \return ierr
    integer function test_tor_fun() result(ierr)
        use num_vars, only: rank
        use grid_utilities, only: calc_eqd_grid
        use dtorh, only: dtorh1

        character(*), parameter :: rout_name = 'test_tor_fun'

        ! local variables
        integer :: jd, kd                                                   ! counters
        integer :: npt                                                      ! number of points
        real(dp) :: lim(2)                                                  ! limits
        real(dp), allocatable :: x(:)                                       ! abscissa
        real(dp), allocatable :: f(:,:,:)                                   ! ordinate P and Q
        integer :: n                                                        ! degree
        integer :: m                                                        ! order
        integer :: max_n                                                    ! max. degree that can be calculated
        integer :: min_n                                                    ! max. degree that can be plot
        logical :: done                                                     ! whether done
        logical :: test_approx                                              ! test approximation
        logical :: test_rel_diff                                            ! relative difference, not absolute
        character(len=max_str_ln), allocatable :: plot_name(:)              ! names of plot
        character(len=max_str_ln), allocatable :: fun_names(:,:)            ! names of functions that are plot

        ! initialize ierr
        ierr = 0

        done = .false.
        test_approx = .false.

        do while (.not.done .and. rank.eq.0)
            call lvl_ud(1)

            ! user input
            call writo('Test approximation at singular point?')
            test_approx = get_log(.false.)
            test_rel_diff = .false.
            if (test_approx) then
                call writo('Relative difference in stead of absolute?')
```

```
            test_rel_diff = get_log(.false.)
        end if

        ! user output
        call writo('Lower bound to plot?')
        lim(1) = get_real(lim_lo=1._dp)
        call writo('Upper bound to plot?')
        lim(2) = get_real(lim_lo=lim(1))
        call writo('How many points?')
        npt = get_int(lim_lo=1)
        call writo('Max. degree? (subscript n in P_(n-0.5)^m)')
        n = get_int(lim_lo=0)
        call writo('Min. degree? (subscript n in P_(n-0.5)^m)')
        min_n = get_int(lim_lo=0,lim_hi=n)
        if (.not.test_approx) then
            call writo('Order? (superscript m in P_(n-0.5)^m)')
            m = get_int(lim_lo=0)
        else
            call writo('Order is 0 when testing approximation')
            m = 0
        end if

        ! set up
        allocate(x(npt))
        allocate(f(npt,0:n,2))
        ierr = calc_eqd_grid(x,lim(1),lim(2))
        chckerr('')
        if (test_approx) then
            allocate(plot_name(min_n:n))
            allocate(fun_names(min_n:n,2))
            if (test_rel_diff) then
                do kd = min_n,n
                    plot_name(kd) = 'tor_Q_rel_diff_'//trim(i2str(kd))
                    fun_names(kd,1) = 'ReldiffQ_{'//trim(i2str(kd))//&
                        &'-0.5}^'//trim(i2str(m))
                    fun_names(kd,2) = 'Q_{'//trim(i2str(kd))//&
                        &'-0.5}^'//trim(i2str(m))
                end do
            else
                do kd = min_n,n
                    plot_name(kd) = 'tor_Q_comp_'//trim(i2str(kd))
                    fun_names(kd,1) = 'ApproxQ_{'//trim(i2str(kd))//&
                        &'-0.5}^'//trim(i2str(m))
                    fun_names(kd,2) = 'Q_{'//trim(i2str(kd))//&
                        &'-0.5}^'//trim(i2str(m))
                end do
            end if
        else
            allocate(plot_name(2))
            allocate(fun_names(2,min_n:n))
            plot_name(1) = 'tor_P'
            plot_name(2) = 'tor_Q'
            do kd = min_n,n
                fun_names(1,kd) = 'P_{'//trim(i2str(kd))//'-0.5}^'//&
                    &trim(i2str(m))
                fun_names(2,kd) = 'Q_{'//trim(i2str(kd))//'-0.5}^'//&
                    &trim(i2str(m))
            end do
        end if

        ! calculate
        do kd = 1,npt
            ierr = dtorh1(x(kd),m,n,f(kd,:,1),f(kd,:,2),max_n)
            chckerr('')
            if (max_n.lt.n) call writo('For point x = '//&
                &trim(r2str(x(kd)))//', the solution did not converge for &
                &degree > '//trim(i2str(max_n)),warning=.true.)
            if (test_approx) then
                f(kd,:,1) = -0.5*log((x(kd)-1._dp)/32)
                do jd = 1,n
                    f(kd,jd:n,1) = f(kd,jd:n,1) - 2_dp/(2._dp*jd-1._dp)
                end do
                if (test_rel_diff) f(kd,:,1) = &
                    &2._dp*(f(kd,:,1)-f(kd,:,2))/(f(kd,:,1)+f(kd,:,2))
            end if
        end do

        ! plot
        do kd = lbound(plot_name,1),ubound(plot_name,1)
            if (test_approx) then
                call print_ex_2d(fun_names(kd,:),trim(plot_name(kd)),&
                    &f(:,kd,:),x=reshape(x,[npt,1]))
            else
                call print_ex_2d(fun_names(kd,:),trim(plot_name(kd)),&
                    &f(:,min_n:n,kd),x=reshape(x,[npt,1]))
            end if
            call draw_ex(fun_names(kd,:),trim(plot_name(kd)),&
```

```
                    &size(fun_names,2),1,.false.)
            end do

            ! clean up
            deallocate(x,f,fun_names,plot_name)

            ! user input
            call writo('Test again?')
            done = .not.get_log(.true.)

            call lvl_ud(-1)
        end do
    end function test_tor_fun

    !> Tests reading of HDF5 subset.
    !!
    !! \return ierr
    integer function test_read_hdf5_subset() result(ierr)
        use num_vars, only: rank, eq_style, n_procs
        use mpi_utilities, only: wait_mpi
        use grid_vars, only: grid_type
        use grid_utilities, only: copy_grid, calc_xyz_grid, trim_grid
        use rich_vars, only: rich_lvl
        use input_utilities, only: dealloc_in
        use pb3d_ops, only: reconstruct_pb3d_in, reconstruct_pb3d_grid, &
            &reconstruct_pb3d_eq_1, reconstruct_pb3d_eq_2, &
            &reconstruct_pb3d_x_1, reconstruct_pb3d_sol
        use eq_vars, only: eq_1_type, eq_2_type
        use x_vars, only: x_1_type, modes_type
        use sol_vars, only: sol_type
        use x_ops, only: init_modes, setup_modes
        use vmec_utilities, only: calc_trigon_factors

        character(*), parameter :: rout_name = 'test_read_HDF5_subset'

        ! local variables
        type(modes_type) :: mds                                         ! general modes variables
        type(grid_type) :: grid                                         ! equilibrium or perturbation grid
        type(grid_type) :: grid_sub                                     ! grid for subset
        type(grid_type) :: grid_trim                                    ! trimmed grid for subset
        type(grid_type) :: grid_eq                                      ! equilibrium grid for XYZ calculation
        type(eq_1_type) :: eq_1                                         ! flux equilibrium variables
        type(eq_2_type) :: eq_2                                         ! metric equilibrium variables
        type(x_1_type) :: X_1                                           ! vectorial perturbation variables
        type(sol_type) :: sol                                           ! solution variables
        integer :: rich_lvl_name                                        ! either the Richardson level or zero, to
    append to names
        integer :: rich_lvl_name_eq                                     ! either the Richardson level or zero, to
    append to names
        integer :: id, jd, kd                                           ! counters
        integer :: i_sub                                                ! counter
        integer :: n_tot(3)                                             ! n of equilibrium grid
        integer :: n_sub(3)                                             ! nr. of subsets
        integer :: n_sub_norm                                           ! nr. of subsets
        integer :: norm_ol = 4                                          ! normal overlap
        integer :: norm_id(2)                                           ! untrimmed normal indices for trimmed grids
        integer :: i_lim(2)                                             ! normal limit for this process
        integer :: plot_dim(3)                                          ! dimensions of plot
        integer :: plot_offset(3)                                       ! local offset of plot
        integer :: test_case                                            ! which case to test (1: eq_2, 2: X_1)
        integer :: i_sec                                                ! which mode for X_1 or sol
        integer, allocatable :: n_sub_loc(:)                            ! number of subsets per process
        integer, allocatable :: n_sub_norm_loc(:)                       ! number of normal poins per process per
    subset
        integer, allocatable :: lims(:,:,:)                             ! limits (3,2,product(n_sub))
        integer, allocatable :: lims_loc(:,:,:)                         ! limits for current process
        real(dp), allocatable :: XYZ(:,:,:,:)                           ! X, Y and Z on output grid
        real(dp), allocatable :: XYZ_i(:,:,:,:)                         ! X, Y and Z for slab plot
        character(len=max_str_ln) :: description                        ! description for plot
        character(len=8) :: name_suff(2)                                ! suffix of name, without and possibly with
    rank appended
        character(len=3) :: grid_name                                   ! either eq or X
        character :: XYZ_name(3)                                        ! X, Y and Z
        logical :: direct_grid_read                                     ! whether grid is directly read with subsets
    or copied
        logical :: divide_over_procs                                    ! whether a division over process should be
    made
        logical :: pers_out                                             ! whether output is persistent (all procs)
    or not (only master)
        logical :: tot_rich                                             ! whether variable is spread out over
    multiple Richardson levels or not

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Checking reading of HDF5 subsets')
```

```
        call lvl_ud(1)

        ! some preliminary things
        ierr = reconstruct_pb3d_in('in')                                    ! reconstruct miscellaneous PB3D output variables
        chckerr('')
        xyz_name(1) = 'X'
        xyz_name(2) = 'Y'
        xyz_name(3) = 'Z'

        ! get information about which variables to reconstruct from user
        call writo('Which variable do you want to test?')
        call lvl_ud(1)
        call writo('1. eq_2: Jac_FD and kappa_n')
        call writo('2. X_1:  IM(U_0), RE(DU_1)')
        call writo('3. sol:  RE(sol_vec)')
        call lvl_ud(-1)
        test_case = get_int(lim_lo=1,lim_hi=3)

        ! set up  whether Richardson level  has to be  appended to the  name and
        ! whether tot_rich is used or not
        select case (eq_style)
            case (1)                                                        ! VMEC
                select case (test_case)
                    case (1)                                                ! eq_2
                        rich_lvl_name = rich_lvl                            ! append richardson level
                        tot_rich = .true.
                    case (2)                                                ! X_1
                        rich_lvl_name = rich_lvl                            ! append richardson level
                        tot_rich = .true.
                    case (3)                                                ! sol
                        rich_lvl_name = 0                                   ! do not append name
                        tot_rich = .false.
                end select
                rich_lvl_name_eq = rich_lvl                                 ! append richardson level
            case (2)                                                        ! HELENA
                rich_lvl_name = 0                                           ! do not append name
                rich_lvl_name_eq = 0                                        ! do not append name
                tot_rich = .false.
        end select
        select case (test_case)
            case (1)                                                        ! eq_2
                grid_name = 'eq'
            case (2)                                                        ! X_1
                grid_name = 'X'
            case (3)                                                        ! sol
                grid_name = 'sol'
        end select

        ! reconstruct full grid
        ierr = reconstruct_pb3d_grid(grid,trim(grid_name),&
            &rich_lvl=rich_lvl_name,tot_rich=.true.)
        chckerr('')
        n_tot = grid%n

        ! get direct read flag from user
        call writo('Read the grids directly using subsets?')
        direct_grid_read = get_log(.true.)

        ! get flag concerning division over MPI procs
        if (n_procs.gt.1) then
            call writo('Divide over MPI processes?')
            divide_over_procs = get_log(.true.)
        else
            divide_over_procs = .false.
        end if

        ! get limits from user
        call writo('Total '//trim(grid_name)//' grid size: ['//&
            &trim(i2str(n_tot(1)))//','//trim(i2str(n_tot(2)))//','//&
            &trim(i2str(n_tot(3)))//']')
        do id = 1,3
            if (n_tot(id).gt.0) then
                call writo('how many divisions in dimension '//trim(i2str(id)))
                n_sub(id) = get_int(lim_lo=0,lim_hi=n_tot(id)-1) + 1
            else
                n_sub(id) = 1
            end if
        end do
        allocate(lims(3,2,product(n_sub)))

        ! also construct equilibrium grid for XYZ calculation, using lims
        lims(1,:,1) = [0,0]
        lims(2,:,1) = [0,0]
        lims(3,:,1) = [-1,-1]
        ierr = reconstruct_pb3d_grid(grid_eq,'eq',rich_lvl=rich_lvl_name_eq,&
            &tot_rich=.true.,lim_pos=lims(:,:,1))
```

```
        chckerr('')
        call print_ex_2d('r_E','r_E_'//trim(i2str(rank)),grid_eq%r_E,&
            &draw=.false.)
        call draw_ex(['r_E'],'r_E_'//trim(i2str(rank)),1,1,.false.)

        ! set limits
        i_sub = 1
        do kd = 1,n_sub(3)
            do jd = 1,n_sub(2)
                do id = 1,n_sub(1)
                    lims(1,1,i_sub) = (id-1)*(n_tot(1)/n_sub(1)) + &
                        &min(mod(n_tot(1),n_sub(1)),id-1) + 1
                    lims(1,2,i_sub) = id*(n_tot(1)/n_sub(1)) + &
                        &min(mod(n_tot(1),n_sub(1)),id)
                    lims(2,1,i_sub) = (jd-1)*(n_tot(2)/n_sub(2)) + &
                        &min(mod(n_tot(2),n_sub(2)),jd-1) + 1
                    lims(2,2,i_sub) = jd*(n_tot(2)/n_sub(2)) + &
                        &min(mod(n_tot(2),n_sub(2)),jd)
                    lims(3,1,i_sub) = (kd-1)*(n_tot(3)/n_sub(3)) + &
                        &min(mod(n_tot(3),n_sub(3)),kd-1) + 1
                    lims(3,2,i_sub) = kd*(n_tot(3)/n_sub(3)) + &
                        &min(mod(n_tot(3),n_sub(3)),kd)
                    i_sub = i_sub+1
                end do
            end do
        end do

        ! divide under processes:  either just dividing over  all processes with
        ! full normal  subranges, or by letting  every process work on  the same
        ! subranges, dividing these normally.
        allocate(n_sub_loc(n_procs))
        allocate(n_sub_norm_loc(n_procs))
        if (divide_over_procs) then
            pers_out = rank.eq.0
            n_sub_loc = product(n_sub)
            allocate(lims_loc(3,2,product(n_sub)))
            lims_loc = lims
            call writo('All process perform every job together, dividing each &
                &one normally with overlap '//trim(i2str(norm_ol)))
        else
            pers_out = .true.
            n_sub_loc = product(n_sub)/n_procs
            n_sub_loc(1:mod(product(n_sub),n_procs)) = &
                &n_sub_loc(1:mod(product(n_sub),n_procs)) + 1
            allocate(lims_loc(3,2,n_sub_loc(rank+1)))
            lims_loc = &
                &lims(:,:,sum(n_sub_loc(1:rank))+1:sum(n_sub_loc(1:rank+1)))
            call writo('Process '//trim(i2str(rank))//' has subsets:',&
                &persistent=.true.)
        end if
        call lvl_ud(1)
        do i_sub = 1,size(lims_loc,3)
            call writo('subset '//trim(i2str(i_sub)),persistent=pers_out)
            call lvl_ud(1)
            do id = 1,3
                call writo('lim('//trim(i2str(id))//'&
                    &) = ['//trim(i2str(lims_loc(id,1,i_sub)))//'..'//&
                    &trim(i2str(lims_loc(id,2,i_sub)))//']',&
                    &persistent=pers_out)
            end do
            call lvl_ud(-1)
        end do
        call lvl_ud(-1)

        ! synchronize MPI
        ierr = wait_mpi()
        chckerr('')

        ! for  X_1 and sol, set  up nm and  get information about which  mode to
        ! plot
        if (test_case.eq.2 .or. test_case.eq.3) then
            ! set up nm in full grids
            ierr = reconstruct_pb3d_eq_1(grid_eq,eq_1,'eq_1')
            chckerr('')
            ierr = init_modes(grid_eq,eq_1)
            chckerr('')
            ierr = setup_modes(mds,grid_eq,grid)
            chckerr('')
            call eq_1%dealloc()
            ! get user input
            call writo('Which perturbation mode do you want to plot?')
            i_sec = get_int(lim_lo=1,lim_hi=size(mds%m,2))
        end if

        ! synchronize MPI
        ierr = wait_mpi()
        chckerr('')
```

```fortran

        ! loop over all josb
        do i_sub = 1,size(lims_loc,3)
            ! user output
            call writo('Process '//trim(i2str(rank))//', subset '//&
                &trim(i2str(i_sub))//':',persistent=.true.)
            call lvl_ud(1)

            ! initialize description
            description = ''
            do id = 1,3
                description = trim(description)//' lim('//trim(i2str(id))//&
                    &') = ['//trim(i2str(lims_loc(id,1,i_sub)))//'..'//&
                    &trim(i2str(lims_loc(id,2,i_sub)))//']'
                if (id.lt.3) description = trim(description)// ','
            end do

            ! set up i_lim
            n_sub_norm = lims_loc(3,2,i_sub)-lims_loc(3,1,i_sub)+1
            if (divide_over_procs) then
                n_sub_norm_loc = n_sub_norm/n_procs
                n_sub_norm_loc(1:mod(n_sub_norm,n_procs)) = &
                    &n_sub_norm_loc(1:mod(n_sub_norm,n_procs)) + 1
                i_lim = [sum(n_sub_norm_loc(1:rank))+1,&
                    &sum(n_sub_norm_loc(1:rank+1))]
                if (rank+1.lt.n_procs) i_lim(2) = i_lim(2)+norm_ol
                description = trim(description)//', normal range = '//&
                    &trim(i2str(i_lim(1)))//'..'//trim(i2str(i_lim(2)))
            else
                n_sub_norm_loc = n_sub_norm
                i_lim = lims_loc(3,:,i_sub) - lims_loc(3,1,i_sub) + 1
            end if

            ! output description
            description = adjustl(description)
            call writo(trim(description),persistent=.true.)

            ! set up subset grid
            if (direct_grid_read) then
                ! check whether it coincides with a direct read using subsets
                ierr = reconstruct_pb3d_grid(grid_sub,trim(grid_name),&
                    &rich_lvl=rich_lvl_name,tot_rich=.true.,&
                    &lim_pos=lims_loc(:,:,i_sub),grid_limits=i_lim)
                chckerr('')
            else
                ierr = copy_grid(grid,grid_sub,&
                    &lims_b=lims_loc(:,:,i_sub),i_lim=i_lim)
                chckerr('')
            end if

            ! trim grid
            ierr = trim_grid(grid_sub,grid_trim,norm_id)
            chckerr('')

            ! set up plot dimensions and local dimensions
            plot_dim = grid_trim%n
            plot_offset = [0,0,grid_trim%i_min-1]

            ! reconstruct metric equilibrium or vectorial perturbation variables
            ! of subset
            select case (test_case)
                case (1)                                                ! eq_2
                    ierr = reconstruct_pb3d_eq_2(grid_sub,eq_2,'eq_2',&
                        &rich_lvl=rich_lvl_name,tot_rich=.true.,&
                        &lim_pos=lims_loc(:,:,i_sub))
                    chckerr('')
                case (2)                                                ! X_1
                    ierr = reconstruct_pb3d_x_1(mds,grid_sub,x_1,'X_1',&
                        &rich_lvl=rich_lvl_name,tot_rich=.true.,&
                        &lim_pos=lims_loc(:,:,i_sub),&
                        &lim_sec_x=[i_sec,i_sec])
                    chckerr('')
                case (3)                                                ! sol
                    ierr = reconstruct_pb3d_sol(mds,grid_sub,sol,'sol',&
                        &rich_lvl=rich_lvl,&
                        &lim_pos=lims_loc(3:3,:,i_sub),&
                        &lim_sec_sol=[i_sec,i_sec])
                    chckerr('')
            end select

            ! set up name suffix
            if (divide_over_procs) then
                name_suff(1) = '_'//trim(i2str(i_sub))
            else
                name_suff(1) = '_'//trim(i2str(sum(n_sub_loc(1:rank))+i_sub))
            end if
            if (divide_over_procs) then
```

```
            name_suff(2) = trim(name_suff(1))//'_'//trim(i2str(rank))
        else
            name_suff(2) = trim(name_suff(1))
        end if

        ! set up XYZ and XYZ_i
        if (test_case.eq.1 .or. test_case.eq.2) then
            if (eq_style.eq.1) then                                     ! if VMEC, calculate trigonometric factors of output grid
                ierr = calc_trigon_factors(grid_trim%theta_E,&
                    &grid_trim%zeta_E,grid_trim%trigon_factors)
                chckerr('')
            end if
            allocate(xyz(grid_trim%n(1),grid_trim%n(2),&
                &grid_trim%loc_n_r,3))
            ierr = calc_xyz_grid(grid_eq,grid_trim,xyz(:,:,:,1),&
                &xyz(:,:,:,2),xyz(:,:,:,3))
            chckerr('')
            allocate(&
                &xyz_i(grid_trim%n(1),grid_trim%n(2),grid_trim%loc_n_r,3))
            do id = 1,grid_trim%n(1)
                xyz_i(id,:,:,1) = lims_loc(1,1,i_sub) + id - 2
            end do
            do jd = 1,grid_trim%n(2)
                xyz_i(:,jd,:,2) = lims_loc(2,1,i_sub) + jd - 2
            end do
            do kd = 1,grid_trim%loc_n_r
                xyz_i(:,:,kd,3) = lims_loc(3,1,i_sub) + kd - 2 + &
                    &grid_trim%i_min - 1
            end do
        else
            allocate(xyz_i(1,grid_trim%loc_n_r,1,1))
            do kd = 1,grid_trim%loc_n_r
                xyz_i(1,kd,1,1) = lims_loc(3,1,i_sub) + kd - 2 + &
                    &grid_trim%i_min - 1
            end do
        end if

        ! plot
        select case (test_case)
            case (1)                                                    ! eq_2
                call plot_hdf5('jac_FD',&
                    &'jac_FD'//trim(name_suff(1)),&
                    &eq_2%jac_FD(:,:,norm_id(1):norm_id(2),0,0,0),&
                    &tot_dim=plot_dim,loc_offset=plot_offset,&
                    &x=xyz(:,:,:,1),y=xyz(:,:,:,2),z=xyz(:,:,:,3),&
                    &descr=description)
                call plot_hdf5('kappa_n',&
                    &'kappa_n'//trim(name_suff(1)),&
                    &eq_2%kappa_n(:,:,norm_id(1):norm_id(2)),&
                    &tot_dim=plot_dim,loc_offset=plot_offset,&
                    &x=xyz(:,:,:,1),y=xyz(:,:,:,2),z=xyz(:,:,:,3),&
                    &descr=description)
            case (2)                                                    ! X_1
                call plot_hdf5('U_0_IM',&
                    &'U_0_IM'//trim(name_suff(1)),&
                    &ip(x_1%U_0(:,:,norm_id(1):norm_id(2),1)),&
                    &tot_dim=plot_dim,loc_offset=plot_offset,&
                    &x=xyz(:,:,:,1),y=xyz(:,:,:,2),z=xyz(:,:,:,3),&
                    &descr=description)
                call plot_hdf5('DU_1_RE',&
                    &'DU_1_RE'//trim(name_suff(1)),&
                    &rp(x_1%DU_1(:,:,norm_id(1):norm_id(2),1)),&
                    &tot_dim=plot_dim,loc_offset=plot_offset,&
                    &x=xyz(:,:,:,1),y=xyz(:,:,:,2),z=xyz(:,:,:,3),&
                    &descr=description)
            case (3)                                                    ! sol
                call print_ex_2d(['sol_RE'],'sol_RE'//trim(name_suff(2)),&
                    &rp(sol%vec(1,norm_id(1):norm_id(2),:)),&
                    &x=xyz_i(1,:,:,1),draw=.false.)
                call draw_ex(['sol_RE'],'sol_RE'//trim(name_suff(2)),&
                    &size(sol%vec,3),1,.false.)
        end select
        if (test_case.eq.1 .or. test_case.eq.2) then
            do id = 1,3
                call plot_hdf5(xyz_name(id),xyz_name(id)//&
                    &trim(name_suff(2)),xyz(:,:,:,id),&
                    &x=xyz_i(:,:,:,1),y=xyz_i(:,:,:,2),z=xyz_i(:,:,:,3))
            end do
        end if

        ! clean up
        call grid_sub%dealloc()
        call grid_trim%dealloc()
        select case (test_case)
            case (1)                                                    ! eq_2
                call eq_2%dealloc()
```

```
                case (2)                                                ! X_1
                    call x_1%dealloc()
                case (3)                                                ! sol
                    call sol%dealloc()
            end select
            if (test_case.eq.1 .or. test_case.eq.2) deallocate(xyz)
            deallocate(xyz_i)
            call lvl_ud(-1)
        end do

        ! synchronize MPI
        ierr = wait_mpi()
        chckerr('')

        ! Now for entire region if wanted
        call writo('Do you want to plot the entire region at once?')
        if (get_log(.false.) .and. rank.eq.0) then
            ! reconstruct metric equilibrium or vectorial perturbation variables
            select case (test_case)
                case (1)                                                ! eq_2
                    ierr = reconstruct_pb3d_eq_2(grid,eq_2,'eq_2',&
                        &rich_lvl=rich_lvl_name,tot_rich=.true.)
                    chckerr('')
                case (2)                                                ! X_1
                    ierr = reconstruct_pb3d_x_1(mds,grid,x_1,'X_1',&
                        &rich_lvl=rich_lvl_name,tot_rich=.true.,&
                        &lim_sec_x=[i_sec,i_sec])
                    chckerr('')
                case (3)                                                ! sol
                    ierr = reconstruct_pb3d_sol(mds,grid,sol,'sol',&
                        &rich_lvl=rich_lvl,&
                        &lim_sec_sol=[i_sec,i_sec])
                    chckerr('')
            end select

            ! set up XYZ and XYZ_i
            if (test_case.eq.1 .or. test_case.eq.2) then
                if (eq_style.eq.1) then                                 ! if VMEC, calculate trigonometric factors
of output grid
                    ierr = calc_trigon_factors(grid%theta_E,&
                        &grid%zeta_E,grid%trigon_factors)
                    chckerr('')
                end if
                allocate(xyz(grid%n(1),grid%n(2),grid%loc_n_r,3))
                ierr = calc_xyz_grid(grid_eq,grid,xyz(:,:,:,1),&
                    &xyz(:,:,:,2),xyz(:,:,:,3))
                chckerr('')
                allocate(xyz_i(grid%n(1),grid%n(2),grid%loc_n_r,3))
                do id = 1,grid%n(1)
                    xyz_i(id,:,:,1) = id - 1
                end do
                do jd = 1,grid%n(2)
                    xyz_i(:,jd,:,2) = jd - 1
                end do
                do kd = 1,grid%loc_n_r
                    xyz_i(:,:,kd,3) = kd - 1
                end do
            else
                allocate(xyz_i(1,grid%loc_n_r,1,1))
                do kd = 1,grid%loc_n_r
                    xyz_i(1,kd,1,1) = kd - 1
                end do
            end if

            ! plot
            description = 'total range'
            select case (test_case)
                case (1)                                                ! eq_2
                    call plot_hdf5('jac_FD','jac_FD_tot',&
                        &eq_2%jac_FD(:,:,:,0,0,0),&
                        &descr=description,&
                        &x=xyz(:,:,:,1),y=xyz(:,:,:,2),z=xyz(:,:,:,3))
                    call plot_hdf5('kappa_n','kappa_n_tot',eq_2%kappa_n,&
                        &descr=description,&
                        &x=xyz(:,:,:,1),y=xyz(:,:,:,2),z=xyz(:,:,:,3))
                case (2)                                                ! X_1
                    call plot_hdf5('U_0_IM','U_0_IM_tot',&
                        &ip(x_1%U_0(:,:,:,1)),&
                        &x=xyz(:,:,:,1),y=xyz(:,:,:,2),z=xyz(:,:,:,3),&
                        &descr=description)
                    call plot_hdf5('DU_1_RE','DU_1_RE_tot',&
                        &rp(x_1%DU_1(:,:,:,1)),&
                        &x=xyz(:,:,:,1),y=xyz(:,:,:,2),z=xyz(:,:,:,3),&
                        &descr=description)
                case (3)                                                ! sol
                    call print_ex_2d(['sol_RE'],'sol_RE_tot',&
                        &rp(sol%vec(1,:,:)),x=xyz_i(1,:,:,1),draw=.false.)
```

```
                    call draw_ex(['sol_RE'],'sol_RE_tot',&
                        &size(sol%vec,3),1,.false.)
            end select
            if (test_case.eq.1 .or. test_case.eq.2) then
                do id = 1,3
                    call plot_hdf5(xyz_name(id),xyz_name(id)//'_tot',&
                        &xyz(:,:,:,id),&
                        &x=xyz_i(:,:,:,1),y=xyz_i(:,:,:,2),z=xyz_i(:,:,:,3))
                end do
            end if

            ! clean up
            call eq_2%dealloc()
            if (test_case.eq.1 .or. test_case.eq.2) deallocate(xyz)
            deallocate(xyz_i)
            call lvl_ud(-1)
        end if

        ! clean up
        call grid%dealloc()
        if (test_case.eq.2 .or. test_case.eq.3) call grid_eq%dealloc()
        call dealloc_in()

        ! synchronize MPI
        ierr = wait_mpi()
        chckerr('')

        ! user output
        call lvl_ud(-1)
        call writo('Test complete')
    end function test_read_hdf5_subset

    !> Tests calculation of volume integral.
    !!
    !! \return ierr
    integer function test_calc_int_vol() result(ierr)
        use num_vars, only: rank
        use grid_utilities, only: calc_eqd_grid, calc_int_vol, &
            &debug_calc_int_vol

        character(*), parameter :: rout_name = 'test_calc_int_vol'

        ! local variables
        real(dp), allocatable :: theta(:,:,:), zeta(:,:,:)                    ! angles
        complex(dp), allocatable :: fun(:,:,:,:)                              ! function to integrate
        real(dp), allocatable :: J(:,:,:)                                     ! Jacobian
        real(dp), allocatable :: X(:,:,:), Y(:,:,:), Z(:,:,:)                 ! X,Y and Z for plot
        real(dp), allocatable :: norm(:)                                      ! normal variable
        real(dp), allocatable :: r(:)                                         ! radius
        real(dp) :: R_0                                                       ! geometric axis of torus
        complex(dp), allocatable :: fun_int(:,:,:)                            ! integrated function
        integer, allocatable :: dims(:,:)                                     ! dimensions of grid
        integer :: n_steps                                                    ! nr. of steps
        integer :: id, jd, kd                                                 ! counters
        real(dp), allocatable :: eqd_grid_dum(:)                              ! equidistant grid dummy
        character(len=max_str_ln) :: var_name(3), file_name(3)                ! name of variable and of file
        complex(dp), allocatable :: fun_int_plot(:,:)                         ! results for plotting

        ! initialize ierr
        ierr = 0

        if (rank.eq.0) then
            call writo('Checking integral of')
            call lvl_ud(1)
            call writo('f = 1 - r^2 + i cos(theta)')
            call lvl_ud(-1)
            call writo('on a torus')

            call writo('Debug mode?')
            if (get_log(.false.)) then
                debug_calc_int_vol = .true.
            else
                debug_calc_int_vol = .false.
            end if

            call writo('How many different grid sizes?')
            n_steps = get_int(lim_lo=1)

            allocate(dims(3,n_steps))
            allocate(eqd_grid_dum(n_steps))

            ! get dimensions
            if (n_steps.gt.1) then
                do kd = 1,size(dims,1)
                    call writo('Starting dimension '//trim(i2str(kd)))
                    dims(kd,1) = get_int(lim_lo=4)
                    call writo('Final dimension '//trim(i2str(kd)))
```

```
                dims(kd,n_steps) = get_int(lim_lo=dims(kd,1)+n_steps)
                ierr = calc_eqd_grid(eqd_grid_dum,1._dp*dims(kd,1),&
                    &1._dp*dims(kd,n_steps))
                chckerr('')
                dims(kd,:) = nint(eqd_grid_dum)
            end do
            call writo('Performing calculations for '//&
                &trim(i2str(n_steps))//' grid sizes')
        else
            do kd = 1,size(dims)
                call writo('Dimension '//trim(i2str(kd)))
                dims(kd,1) = get_int(lim_lo=1)
                if (dims(kd,1).eq.1) then
                    call lvl_ud(1)
                    call writo('Make sure the integrand does not depend on &
                        &dimension '//trim(i2str(kd)))
                    call writo('A factor such as 2pi can be missing from &
                        &resulting integral')
                    call lvl_ud(-1)
                end if
            end do
        end if

        allocate(fun_int(2,1,n_steps))

        do jd = 1,n_steps
            ! user output
            call writo('grid size = ['//trim(i2str(dims(1,jd)))//' ,'//&
                &trim(i2str(dims(2,jd)))//', '//trim(i2str(dims(3,jd)))//&
                &']')
            call lvl_ud(1)

            ! set up variables
            allocate(theta(dims(1,jd),dims(2,jd),dims(3,jd)))
            allocate(zeta(dims(1,jd),dims(2,jd),dims(3,jd)))
            allocate(j(dims(1,jd),dims(2,jd),dims(3,jd)))
            allocate(fun(dims(1,jd),dims(2,jd),dims(3,jd),1))
            allocate(x(dims(1,jd),dims(2,jd),dims(3,jd)))
            allocate(y(dims(1,jd),dims(2,jd),dims(3,jd)))
            allocate(z(dims(1,jd),dims(2,jd),dims(3,jd)))
            allocate(norm(dims(3,jd)))
            allocate(r(dims(3,jd)))

            r_0 = 2._dp
            ierr = calc_eqd_grid(theta,0*pi,2*pi,1)
            chckerr('')
            ierr = calc_eqd_grid(zeta,0*pi,2*pi,2)
            chckerr('')
            ierr = calc_eqd_grid(norm,0._dp,1._dp)
            chckerr('')
            do kd = 1,dims(3,jd)
                r(kd) = (kd-1._dp)/(dims(3,jd)-1)
                fun(:,:,kd,1) = 1._dp - r(kd)**2 + iu*cos(theta(:,:,kd))
                do id = 1,dims(1,jd)
                    j(id,:,kd) = r(kd)*(r_0+r(kd)*cos(theta(id,:,kd)))
                end do
                x(:,:,kd) = cos(zeta(:,:,kd))*(r_0+r(kd)*cos(theta(:,:,kd)))
                y(:,:,kd) = sin(zeta(:,:,kd))*(r_0+r(kd)*cos(theta(:,:,kd)))
                z(:,:,kd) = r(kd)*sin(theta(:,:,kd))
            end do

            var_name(1) = 'TEST_J_torus'
            var_name(2) = 'TEST_RE_fun_torus'
            var_name(3) = 'TEST_IM_fun_torus'
            file_name(1) = 'TEST_J_torus'
            file_name(2) = 'TEST_RE_fun_torus'
            file_name(3) = 'TEST_IM_fun_torus'
            if (n_steps.gt.1) then
                do kd = 1,3
                    file_name(kd) = trim(file_name(kd))//'_'//trim(i2str(jd))
                end do
            end if
            call plot_hdf5(var_name(1),file_name(1),j,x=x,y=y,z=z)
            call plot_hdf5(var_name(2),file_name(2),rp(fun(:,:,:,1)),&
                &x=x,y=y,z=z)
            call plot_hdf5(var_name(3),file_name(3),ip(fun(:,:,:,1)),&
                &x=x,y=y,z=z)

            ! user output
            call writo('Analytically calculating integral')

            ! calculate integral analytically
            fun_int(1,1,jd) = r_0*pi**2 + iu*2*pi**2/3

            ! user output
            call writo('Numerically calculating integral')

```

```
                ! calculate integral numerically
                ierr = calc_int_vol(theta,zeta,norm,j,fun,fun_int(2,:,jd))
                chckerr('')

                ! output
                call writo('Analytical value: '//&
                    &trim(c2str(fun_int(1,1,jd))))
                call writo('Numerical value: '//&
                    &trim(c2str(fun_int(2,1,jd))))

                ! clean up
                deallocate(theta,zeta,j,fun)
                deallocate(x,y,z)
                deallocate(norm,r)

                call lvl_ud(-1)
            end do

            if (n_steps.gt.1) then
                allocate(fun_int_plot(n_steps,2))
                do jd = 1,n_steps
                    fun_int_plot(jd,:) = fun_int(:,1,jd)/fun_int(1,1,jd)
                end do
                call print_ex_2d(['analytical integral, RE',&
                    &'numerical integral, RE '],'',rp(fun_int_plot))
                call print_ex_2d(['analytical integral, IM',&
                    &'numerical integral, IM '],'',ip(fun_int_plot))
            end if
        end if
    end function test_calc_int_vol
#endif
end module test
```

D.86 /Users/toonweyens/Code/PB3D/Modules/vac_ops.f90 File Reference

Modules

- module vac_ops

  Operations and variables pertaining to the vacuum response.

Functions/Subroutines

- integer function, public vac_ops::store_vac (grid, eq_1, eq_2, vac)

  Stores calculation of the vacuum response by storing the necessary variables.
- integer function vac_ops::calc_gh_1 (vac, n_r_in, lims_r_in, x_vec_in, g_in, h_in, ext_in)

  Calculates matrices `G` and `H` in 3-D configuration.
- logical function on_field_line (cd, kd, n_ang)
- integer function vac_ops::calc_gh_2 (vac, n_r_in, lims_r_in, x_vec_in, g_in, h_in, ext_in)

  Calculates matrices `G` and `H` in axisymmetric configurations.
- integer function, public vac_ops::calc_vac_res (mds, vac)

  Calculates the vacuum response.
- integer function, public vac_ops::vac_pot_plot (grid_sol, vac, sol, x_id)

  Calculate vacuum potential at some positions that are not resonant.
- integer function vac_ops::solve_phi_bem (vac, r, phi, n_rphi, n_loc_rphi, lims_c_rphi, desc_rphi)

  Calculates potential `Phi` on boundary of BEM in terms of some right-hand side .
- integer function, public vac_ops::print_output_vac (vac, data_name, rich_lvl)

  Print vacuum quantities of a certain order to an output file.

Variables

- logical, public vac_ops::debug_calc_gh = .false.

  plot debug information for calc_GH()
- logical, public vac_ops::debug_calc_vac_res = .false.

  plot debug information for calc_vac_res()
- logical, public vac_ops::debug_vac_pot_plot = .false.

  plot debug information for vac_pot_plot()

D.86.1 Function/Subroutine Documentation

D.86.1.1 on_field_line()

```
logical function calc_gh_1::on_field_line (
                integer, intent(in) cd,
                integer, intent(in) kd,
                integer, intent(in) n_ang)
```

Definition at line 964 of file vac_ops.f90.

Here is the caller graph for this function:

D.87 vac_ops.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Operations and variables pertaining to the vacuum response.
!!
!! The vacuum  response is calculated  using the Boundary Element  Method. There
!! are two possibilities, indicated by the variable \c style in \c vac_vars.
!!  -# 3-D field aligned:
!!      - Makes  use of  the collocation  method for the  grid points  along the
!!      magnetic field lines.
!!      - The 3-D integrals are therefore integrated using Weyl's theorem
!!      \cite Helander2014, p. 5.
!!  -# axisymmetric:
!!      - Makes use of an analytical  expression for the toroidal integration of
!!      the Green's functions, as shown in \cite Cohl1999.
!!      - The  integration in the poloidal  angle is done using  the collocation
!!      method.
!!
!! The final matrix equation is solved through Strumpack \cite Meiser2016.
!!
!! \see See \cite Weyens3D.
!!
!! \todo The vacuum part of PB3D is still under construction and not usable yet.
!-----------------------------------------------------------------------------!
module vac_ops
#include <PB3D_macros.h>
#include <wrappers.h>
    use strumpackdensepackage
    use str_utilities
    use messages
    use output_ops
```

```
    use num_vars, only: dp, pi, max_str_ln, iu
    use grid_vars, only: grid_type
    use eq_vars, only: eq_1_type, eq_2_type
    use x_vars, only: x_2_type, modes_type
    use vac_vars, only: copy_vac, &
        &vac_type

    implicit none
    private
    public store_vac, calc_vac_res, print_output_vac, vac_pot_plot
#if ldebug
    public debug_calc_gh, debug_calc_vac_res, debug_vac_pot_plot
#endif

    ! global variables
#if ldebug
    logical :: debug_calc_gh = .false.                                  !< plot debug information for calc_GH()
\ldebug
    logical :: debug_calc_vac_res = .false.                             !< plot debug information for calc_vac_res()
\ldebug
    logical :: debug_vac_pot_plot = .false.                             !< plot debug information for vac_pot_plot()
\ldebug
#endif

contains
    !> Stores  calculation  of the  vacuum  response  by storing  the  necessary
    !! variables.
    !!
    !! This is  done by  the last process,  which is the  one that  contains the
    !! variables at the  plasma edge, and then this is  broadcasted to the other
    !! processes.
    !!
    !! The workings of this routine  are summarized in the following diagram for
    !! VMEC:
    !! @dot
    !! digraph G {
    !!  eq_job -> restart [label=yes]
    !!  eq_job -> store [label=no]
    !!  restart -> jump [label=yes]
    !!  restart -> copy [label=no]
    !!  jump -> reconstructcur [label=yes]
    !!  jump -> restartone [label=no]
    !!  reconstructcur -> return
    !!  restartone -> reconstruct [label=no]
    !!  copy -> store
    !!  reconstruct -> store
    !!  restartone -> store [label=yes]
    !!  store -> return
    !!
    !!  eq_job [shape=diamond,label=eq_job_1]
    !!  store [label=store_new_vec_res]
    !!  restart [shape=diamond,label=restart]
    !!  jump [shape=diamond,label=jump]
    !!  copy [label=copy_prev_level]
    !!  reconstructcur [label=reconstruct_cur_level]
    !!  return [label=return]
    !!  restartone [shape=diamond,label=restart_1]
    !!  reconstruct [label=reconstruct_prev_level]
    !! }
    !! @enddot
    !!
    !! Before  storing the  new variables,  the procedure  checks the  following
    !! things:
    !!  - For  the first equilibrium job,  this routine copies the  results from
    !!  the previous Richardson level into  the appropriate subranges of the new
    !!  vacuum variables if no restart of Richardson levels was done.
    !!  - If a  restart was done and the  level is greater than one,  there is a
    !!  reconstruction of the  previous level's variables. But  this will happen
    !!  in calc_vac_res(), not in this procedure.
    !!
    !! If  there is  a jump  straight to  the solution,  however, the  procedure
    !! returns after reconstructing the current level's variables.
    !!
    !! If the previous \c  G and \c H variables are  empty, they are regenerated
    !! later,  in  calc_gh(). This  indicates  that  a reconstruction  happened,
    !! either  because a  Richardson restart  was  performed, or  because a  new
    !! Richardson  level was  started after  a previous  level in  which it  was
    !! jumped straight to the solution.
    !!
    !! For  HELENA, there  is only  1 equilibrium  job at  the first  Richardson
    !! level, and the vacuum has to be  calculated only once then. If there is a
    !! jump to solution or  if there is Richardson restart, it  only needs to be
    !! reconstructed.
    !!
    !! \return ierr
    integer function store_vac(grid,eq_1,eq_2,vac) result(ierr)
        use pb3d_ops, only: reconstruct_pb3d_vac
```

```fortran
        use num_vars, only: rich_restart_lvl, eq_job_nr, eq_jobs_lims, &
            &eq_style, rank, n_procs, use_pol_flux_f
        use rich_vars, only: rich_lvl
        use mpi_utilities, only: broadcast_var
        use num_utilities, only: c
        use grid_vars, only: n_r_eq
        use x_vars, only: prim_x

        character(*), parameter :: rout_name = 'store_vac'

        ! input / output
        type(grid_type), intent(in) :: grid                                     !< equilibrium grid
        type(eq_1_type), intent(in) :: eq_1                                     !< flux equilibrium variables
        type(eq_2_type), intent(in) :: eq_2                                     !< metric equilibrium variables
        type(vac_type), intent(inout) :: vac                                    !< vacuum variables

        ! local variables
        integer :: id, jd                                                       ! counter
        integer :: r_id                                                         ! normal index
        real(dp) :: jq                                                          ! iota or q
        character(len=max_str_ln) :: err_msg                                    ! error message

        ! initialize ierr
        ierr = 0

        ! set jq
        if (use_pol_flux_f) then
            jq = eq_1%q_saf_FD(grid%loc_n_r,0)
        else
            jq = eq_1%rot_t_FD(grid%loc_n_r,0)
            ierr = 1
            err_msg = 'TOR. FLUX HAS NOT BEEN TESTED AND IS MOST PROBABLY NOT &
                &CORRECT YET'
            chckerr(err_msg)
        end if
        ierr = broadcast_var(jq,n_procs-1)
        chckerr('')

        ! call the approriate procedure
        select case (eq_style)
            case (1)                                                            ! VMEC
                ierr = store_vac_vmec()
                chckerr('')
            case (2)                                                            ! HELENA
                ierr = store_vac_hel()
                chckerr('')
        end select
    contains
        ! VMEC version
        !> \private
        integer function store_vac_vmec() result(ierr)
            use num_vars, only: jump_to_sol
            use grid_vars, only: n_alpha
            use rich_vars, only: n_par_x
            use eq_utilities, only: calc_inv_met
            use vac_utilities, only: interlaced_vac_copy

            character(*), parameter :: rout_name = 'store_vac_VMEC'

            ! local variables
            integer :: par_id(3)                                                ! total parallel index
            real(dp), allocatable :: norm_com_c(:,:,:)                          ! Cylindrical components of norm
            real(dp), allocatable :: t_cv_loc(:,:,:,:,:,:,:)                    ! local T_CV
            type(vac_type) :: vac_old                                           ! old vacuum variables
            logical :: interlaced_vac_copy_needed                               ! interlaced copy vacuum needed

            ! initialize ierr
            ierr = 0

            ! user output
            call writo('Start storing vacuum quantities')

            call lvl_ud(1)

            ! if start of Richardson level  that is not the first, copy previous
            ! results
            if (eq_job_nr.eq.1) then
                if (rich_lvl.eq.rich_restart_lvl) then                          ! restarting
                    if (jump_to_sol) then                                       ! jumping straight to solution
                        ! nothing needs to be done here
                        return
                    else                                                        ! not jumping to solution
                        if (rich_restart_lvl.eq.1) then                         ! not actually restarting, just starting
                            ! nothing extra needed
                            interlaced_vac_copy_needed = .false.
                        else                                                    ! restarting
                            ! reconstruct old vacuum
```

```
                        ierr = reconstruct_pb3d_vac(vac_old,'vac',&
                            &rich_lvl=rich_lvl-1)
                        chckerr('')

                        ! interlaced copy necessary
                        interlaced_vac_copy_needed = .true.
                    end if
                end if
            else                                                        ! not restarting
                ! copy old vacuum temporarily
                ierr = copy_vac(vac,vac_old)
                chckerr('')

                ! deallocate
                call vac%dealloc()

                ! interlaced copy necessary
                interlaced_vac_copy_needed = .true.
            end if

            ! allocate
            ierr = vac%init(1,n_par_x*n_alpha,prim_x,[n_par_x,n_alpha],jq)
            chckerr('')

            ! interlaced copy if needed
            if (interlaced_vac_copy_needed) then
                ierr = interlaced_vac_copy(vac_old,vac)
                chckerr('')
            end if
        end if

        ! add results from current equilibrium job to the variables
        if (rank.eq.n_procs-1) then
            ! grid point to save is last local
            r_id = grid%loc_n_r

            ! calculate Cartesian components of J nabla psi
            !   = J_F (Dpsi_pol/Dr_V) / 2pi (T_C^V)^T (e^C)
            !   = J_F T_F^V(1,2) (T_C^V)^T (e^C)
            allocate(norm_com_c(grid%n(1),grid%n(2),3))
            norm_com_c = 0._dp
            allocate(t_cv_loc(size(eq_2%T_VC,1),size(eq_2%T_VC,2),1:1,&
                &size(eq_2%T_VC,4),0:0,0:0,0:0))
            ierr = calc_inv_met(t_cv_loc,eq_2%T_VC(:,:,r_id:r_id,:,:,:,:),&
                &[0,0,0])
            chckerr('')
            do id = 1,3
                norm_com_c(:,:,id) = eq_2%jac_FD(:,:,r_id,0,0,0) * &
                    &eq_2%T_EF(:,:,r_id,c([1,2],.false.),0,0,0)*&
                    &t_cv_loc(:,:,1,c([id,1],.false.),0,0,0)            ! Cylindrical contravariant components
            end do
            deallocate(t_cv_loc)

            ! iterate over all field lines
            do jd = 1,n_alpha
                ! set total parallel id
                if (rich_lvl.eq.1) then
                    par_id = [eq_jobs_lims(:,eq_job_nr),1]
                else
                    par_id = [2*eq_jobs_lims(:,eq_job_nr),2]
                end if
                par_id(1:2) = par_id(1:2) + (jd-1)*n_par_x

                ! save X, Y and Z
                vac%x_vec(par_id(1):par_id(2):par_id(3),1) = &
                    &eq_2%R_E(:,jd,r_id,0,0,0) * cos(grid%zeta_E(:,jd,r_id))
                vac%x_vec(par_id(1):par_id(2):par_id(3),2) = &
                    &eq_2%R_E(:,jd,r_id,0,0,0) * sin(grid%zeta_E(:,jd,r_id))
                vac%x_vec(par_id(1):par_id(2):par_id(3),3) = &
                    &eq_2%Z_E(:,jd,r_id,0,0,0)

                ! save norm
                vac%norm(par_id(1):par_id(2):par_id(3),1) = &
                    &norm_com_c(:,jd,1)*cos(grid%zeta_E(:,jd,r_id)) - &
                    &norm_com_c(:,jd,2)*sin(grid%zeta_E(:,jd,r_id)) / &
                    &eq_2%R_E(:,jd,r_id,0,0,0)                          ! ~ e^X
                vac%norm(par_id(1):par_id(2):par_id(3),2) = &
                    &norm_com_c(:,jd,1)*sin(grid%zeta_E(:,jd,r_id)) + &
                    &norm_com_c(:,jd,2)*cos(grid%zeta_E(:,jd,r_id)) / &
                    &eq_2%R_E(:,jd,r_id,0,0,0)                          ! ~ e^X
                vac%norm(par_id(1):par_id(2):par_id(3),3) = &
                    &norm_com_c(:,jd,3)                                 ! ~ e^Z

                ! save metric factors and H factor
                vac%h_fac(par_id(1):par_id(2):par_id(3),1) = &
                    &eq_2%g_FD(:,jd,r_id,c([1,1],.true.),0,0,0)         ! g_11
                vac%h_fac(par_id(1):par_id(2):par_id(3),2) = &
```

```
                    &eq_2%g_FD(:,jd,r_id,c([1,3],.true.),0,0,0)             ! g_13
                vac%h_fac(par_id(1):par_id(2):par_id(3),3) = &
                    &eq_2%g_FD(:,jd,r_id,c([3,3],.true.),0,0,0)             ! g_33
                vac%h_fac(par_id(1):par_id(2):par_id(3),4) = 0.5_dp* &
                    &eq_2%jac_FD(:,jd,r_id,0,0,0)**2 * &
                    &(eq_2%h_FD(:,jd,r_id,c([2,2],.true.),0,0,0) / &
                    &eq_2%g_FD(:,jd,r_id,c([1,1],.true.),0,0,0))**1.5 * ( &
                    &eq_2%jac_FD(:,jd,r_id,0,1,0)/&
                    &eq_2%jac_FD(:,jd,r_id,0,0,0) + &
                    &0.5_dp*eq_2%h_FD(:,jd,r_id,c([2,2],.true.),0,1,0)/&
                    &eq_2%h_FD(:,jd,r_id,c([2,2],.true.),0,0,0) - &
                    &0.5_dp*eq_2%g_FD(:,jd,r_id,c([1,1],.true.),0,1,0)/&
                    &eq_2%g_FD(:,jd,r_id,c([1,1],.true.),0,0,0) &
                    &)                                                      ! J/2 h^22/g_11 d/d2 (J sqrt(h^22/g_11)))
            end do
            !call print_ex_2D(['norm'],'norm_'//trim(i2str(rich_lvl)),&
                !&vac%norm,persistent=.true.)
            !call print_ex_2D(['x_vec'],'x_vec'//trim(i2str(rich_lvl)),&
                !&vac%x_vec,persistent=.true.)
        end if

        ! broadcast to other processes
        do id = 1,size(vac%x_vec,2)
            ierr = broadcast_var(vac%x_vec(:,id),n_procs-1)
            chckerr('')
            ierr = broadcast_var(vac%norm(:,id),n_procs-1)
            chckerr('')
        end do
        do id = 1,size(vac%h_fac,2)
            ierr = broadcast_var(vac%h_fac(:,id),n_procs-1)
            chckerr('')
        end do

        call lvl_ud(-1)

        call writo('Done storing vacuum quantities')
    end function store_vac_vmec
    ! HELENA version
    !> \private
    integer function store_vac_hel() result(ierr)
        use helena_vars, only: r_h, z_h, nchi, chi_h, ias
        use grid_utilities, only: nufft

        character(*), parameter :: rout_name = 'store_vac_HEL'

        ! local variables
        integer :: n_theta                                                  ! total number of points, including overlap
        real(dp), allocatable :: x_vec_f(:,:)                               ! Fourier components of x_vec
        character(len=max_str_ln) :: err_msg                                ! error message

        ! initialize ierr
        ierr = 0

        ierr = 2
        chckerr('Vacuum has not been implemented yet!')

        ! user output
        call writo('Start storing vacuum quantities')

        call lvl_ud(1)

        ! if  restarting  a  Richardson  level, reconstruct.  If  not,  only
        ! calculate for the first level
        if (rich_lvl.eq.rich_restart_lvl) then
            ! check for start from zero versus restart
            if (rich_restart_lvl.eq.1) then                                 ! starting from zero
                ! test
                if (eq_job_nr.ne.1) then
                    ierr = 1
                    err_msg = 'HELENA should only have 1 equilibrium job &
                        &for Richardson level 1'
                    chckerr(err_msg)
                end if

                if (ias.eq.0) then                                          ! top-bottom symmmetric
                    n_theta = 2*(nchi-1)+1
                else
                    n_theta = nchi
                end if

                ! allocate
                ierr = vac%init(eq_style,n_theta,prim_x,[n_theta,1],jq)
                chckerr('')
                ! save points
                if (rank.eq.n_procs-1) then
                    ! grid point to save is last total
                    r_id = n_r_eq
```

```
                    ! save only R and Z
                    if (ias.eq.0) then
                        vac%x_vec(1:nchi,1) = r_h(1:nchi,r_id)
                        vac%x_vec(1:nchi,2) = z_h(1:nchi,r_id)
                        vac%ang(1:nchi,1) = chi_h(1:nchi)
                        vac%x_vec(nchi+1:n_theta,1) = r_h(nchi-1:1:-1,r_id)
                        vac%x_vec(nchi+1:n_theta,2) = -z_h(nchi-1:1:-1,r_id)
                        vac%ang(nchi+1:n_theta,1) = chi_h(2:nchi) + pi
                    else
                        vac%x_vec(:,1) = r_h(:,r_id)
                        vac%x_vec(:,2) = z_h(:,r_id)
                        vac%ang(:,1) = chi_h
                    end if
                    !call print_ex_2D(['R','Z'],'',vac%x_vec,&
                        !&x=vac%ang(:,1:1),persistent=.true.)
                    !call print_ex_2D('cs','',vac%x_vec(:,2),&
                        !&x=vac%x_vec(:,1),persistent=.true.)

                    ! calculate Cylindrical components of - J nabla psi
                    !   = - R (Z_theta nabla R - R_theta nabla Z)
                    ! and its poloidal derivative
                    vac%norm = 0._dp
                    vac%dnorm = 0._dp
                    ! nufft of Z
                    ierr = nufft(vac%ang(1:n_theta-1,1),&
                        &vac%x_vec(1:n_theta-1,2),x_vec_f)
                    chckerr('')
                    ! dZ/dtheta
                    do id = 1,size(x_vec_f,1)
                        vac%norm(1:n_theta-1,1) = &
                            &vac%norm(1:n_theta-1,1) + (id-1) * &
                            &(-x_vec_f(id,1)*sin((id-1)*&
                            &vac%ang(1:n_theta-1,1))&
                            &+x_vec_f(id,2)*cos((id-1)*&
                            &vac%ang(1:n_theta-1,1)))
                    end do
                    ! nufft of -R
                    ierr = nufft(vac%ang(1:n_theta-1,1),&
                        &-vac%x_vec(1:n_theta-1,1),x_vec_f)
                    chckerr('')
                    ! d(-R)/dtheta
                    do id = 1,size(x_vec_f,1)
                        vac%norm(1:n_theta-1,2) = &
                            &vac%norm(1:n_theta-1,2) + (id-1) * &
                            &(-x_vec_f(id,1)*sin((id-1)*&
                            &vac%ang(1:n_theta-1,1))&
                            &+x_vec_f(id,2)*cos((id-1)*&
                            &vac%ang(1:n_theta-1,1)))
                    end do
                    vac%norm(n_theta,:) = vac%norm(1,:)
                    ! multiply by -R
                    do id = 1,2
                        vac%norm(:,id) = - vac%x_vec(:,1)*vac%norm(:,id)
                    end do
                    ! nufft of norm
                    do jd = 1,2
                        ierr = nufft(vac%ang(1:n_theta-1,1),&
                            &vac%norm(1:n_theta-1,jd),x_vec_f)
                        chckerr('')
                        do id = 1,size(x_vec_f,1)
                            vac%dnorm(1:n_theta-1,jd) = &
                                &vac%dnorm(1:n_theta-1,jd) + (id-1) * &
                                &(-x_vec_f(id,1)*sin((id-1)*&
                                &vac%ang(1:n_theta-1,1))&
                                &+x_vec_f(id,2)*cos((id-1)*&
                                &vac%ang(1:n_theta-1,1)))
                        end do
                    end do
                    vac%dnorm(n_theta,:) = vac%dnorm(1,:)
                end if

                ! broadcast to other processes
                do id = 1,2
                    ierr = broadcast_var(vac%x_vec(:,id),n_procs-1)
                    chckerr('')
                    ierr = broadcast_var(vac%norm(:,id),n_procs-1)
                    chckerr('')
                    ierr = broadcast_var(vac%dnorm(:,id),n_procs-1)
                    chckerr('')
                end do
                ierr = broadcast_var(vac%ang(:,1),n_procs-1)
                chckerr('')
            else                                                    ! restarting
                ! reconstruct old vacuum
                ierr = reconstruct_pb3d_vac(vac,'vac')
                chckerr('')
```

```
                end if
            end if

            call lvl_ud(-1)

            call writo('Done storing vacuum quantities')
        end function store_vac_hel
    end function store_vac

    !> Calculate the matrices \c G and \c H.
    !!
    !! This  is  done  by  iterating  over  the  subintervals,  calculating  the
    !! contributions to the  potential values on the left and  right boundary of
    !! the interval, and adding these to the appropriate values in G and H.
    !!
    !! As each process in  the blacs grid only has access to  its own values, an
    !! extra interval  is is calculated  to the left  and right of  the internal
    !! process grid edges.
    !!
    !! Optionally, this  procedure can be  used to calculate the  coefficients G
    !! and H  with respect  to other  points, outside of  the boundary.  This is
    !! useful when the potential is to be calculated at external points.
    !!
    !! In this case, however, no (near-)  singular points are calculated, and if
    !! they appear any way, perhaps by accident, they will not be accurate.
    !!
    !! \Note Multiple  field lines are  stored sequentially, which  implies that
    !! the integral between the  last point on a field line  and the first point
    !! on the next field lines needs to be left out.
    !!
    !! \todo For  3-D vacuums,  step sizes  are constant.  Subsequent Richardson
    !! levels should therefore make use of the fact that the contribution to the
    !! points inherited from  the previous levels can be just  scaled by 1/2 and
    !! do not need to be recalculated.  Currently, the copy is done correctly in
    !! interlaced_vac_copy(), but they are afterwards overwritten.
    !!
    !! \return ierr
    integer function calc_gh(vac,n_r_in,lims_r_in,x_vec_in,G_in,H_in) &
        &result(ierr)

        use num_vars, only: rank
        use vac_vars, only: in_context
        use vac_utilities, only: vec_dis2loc
#if ldebug
        use num_vars, only: n_procs
        use vac_utilities, only: mat_dis2loc
#endif

        character(*), parameter :: rout_name = 'calc_GH'

        ! input / output
        type(vac_type), intent(inout), target :: vac                          !< vacuum variables
        integer, intent(in), optional :: n_r_in                               !< total number of rows of external poins of influence
        integer, intent(in), optional, target :: lims_r_in(:,:)               !< row limits of external points of influence
        real(dp), intent(in), optional, target :: x_vec_in(:,:)               !< external position of influence
        real(dp), intent(in), optional, target :: g_in(:,:)                   !< external G
        real(dp), intent(in), optional, target :: h_in(:,:)                   !< external H

        ! local variables
        integer :: n_r                                                        ! number of rows
        integer :: id, jd                                                     ! counters
        integer, pointer :: lims_r(:,:)                                       ! row limits
        real(dp), pointer :: g(:,:)                                           ! G
        real(dp), pointer :: h(:,:)                                           ! H
        real(dp), pointer :: x_vec(:,:)                                       ! x_vec used in row
        character(len=max_str_ln) :: err_msg                                  ! error message
        logical :: ext_in                                                     ! external influence point
#if ldebug
        integer :: rd                                                         ! global counter for row
        integer :: rdl                                                        ! local counter for row
        integer :: i_rd                                                       ! index of subrow
        integer :: desc_res(blacsctxtsize)                                    ! descriptor for the result
        real(dp), allocatable :: loc_ser(:,:)                                 ! dummy serial variable
        real(dp), allocatable :: r_sph(:)                                     ! spherical r with r^2 = R^2+Z^2
        real(dp), allocatable :: phi(:,:)                                     ! test phi
        real(dp), allocatable :: dphi(:,:)                                    ! test dphi/dn
        real(dp), allocatable :: res(:,:,:)                                   ! H phi, G dphi
        character(len=max_str_ln) :: plot_title(3)                            ! plot title
        character(len=max_str_ln) :: plot_name                                ! plot name
#endif

        ! initialize ierr
        ierr = 0

        ! user output
```

```
        call writo('Calculate the G and H matrices')
        call lvl_ud(1)

        ! test
        if (present(n_r_in) .and. .not.present(x_vec_in) .or. &
            &present(n_r_in) .and. .not.present(lims_r_in) .or. &
            &present(n_r_in) .and. .not.present(g_in) .or. &
            &present(n_r_in) .and. .not.present(h_in)) then
            ierr = 1
            chckerr('Need all optional variables')
        end if

        ! set limits and variables
        if (present(lims_r_in)) then
            n_r = n_r_in
            lims_r => lims_r_in
            x_vec => x_vec_in
            g => g_in
            h => h_in
            ext_in = .true.
        else
            n_r = vac%n_bnd
            allocate(lims_r(2,size(vac%lims_r,2)))
            allocate(x_vec(vac%n_bnd,size(vac%x_vec,2)))
            lims_r = vac%lims_r
            x_vec = vac%x_vec
            g => vac%G
            h => vac%H
            ext_in = .false.
        end if

        ! call specific procedure for vacuum style
        select case (vac%style)
            case (1)                                                        ! field-line 3-D
                ierr = calc_gh_1(vac,n_r,lims_r,x_vec,g,h,ext_in)
                chckerr('')
            case (2)                                                        ! axisymmetric
                ierr = calc_gh_2(vac,n_r,lims_r,x_vec,g,h,ext_in)
                chckerr('')
            case default
                ierr = 1
                err_msg = 'No vacuum style '//trim(i2str(vac%style))//&
                    &' possible'
                chckerr(err_msg)
        end select

        ! clean up
        if (.not.present(lims_r_in)) then
            deallocate(lims_r)
            deallocate(x_vec)
        end if
        nullify(lims_r)
        nullify(x_vec)
        nullify(g)
        nullify(h)

#if ldebug
        if (debug_calc_gh .and. in_context(vac%ctxt_HG)) then
            ! initialize
            call descinit(desc_res,vac%n_bnd,1,vac%bs,vac%bs,0,0,&
                &vac%ctxt_HG,max(1,vac%n_loc(1)),ierr)
            chckerr('descinit failed for res')

            ! plot H and G in HDF5
            allocate(loc_ser(vac%n_bnd,vac%n_bnd))
            ierr = mat_dis2loc(vac%ctxt_HG,vac%H,vac%lims_r,vac%lims_c,loc_ser,&
                &proc=n_procs-1)
            chckerr('')
            if (rank.eq.n_procs-1) then
                call plot_hdf5('H','H',reshape(loc_ser,[vac%n_bnd,vac%n_bnd,1]))
                call plot_diff_hdf5(reshape(loc_ser,[vac%n_bnd,vac%n_bnd,1]),&
                    &reshape(transpose(loc_ser),[vac%n_bnd,vac%n_bnd,1]),&
                    &'H_transp')
            end if
            ierr = mat_dis2loc(vac%ctxt_HG,vac%G,vac%lims_r,vac%lims_c,loc_ser,&
                &proc=n_procs-1)
            chckerr('')
            if (rank.eq.n_procs-1) then
                call plot_hdf5('G','G',reshape(loc_ser,[vac%n_bnd,vac%n_bnd,1]))
                call plot_diff_hdf5(reshape(loc_ser,[vac%n_bnd,vac%n_bnd,1]),&
                    &reshape(transpose(loc_ser),[vac%n_bnd,vac%n_bnd,1]),&
                    &'G_transp')
            end if
            deallocate(loc_ser)

            ! the rest is only for axisymmetric vacua.
            if (vac%style.eq.2) then
```

```
            ! test whether G and H hold for test potentials phi
            if (vac%lims_c(1,1).eq.1) then                                  ! this process owns (part of) first column
                ! set up variables
                allocate(res(vac%n_loc(1),3,2))
                allocate(r_sph(vac%n_loc(1)))
                allocate(phi(vac%n_loc(1),2))
                allocate(dphi(vac%n_loc(1),2))

                ! set up rhs for H (phi) and G (dphi)
                subrows: do i_rd = 1,size(vac%lims_r,2)
                    row: do rd = vac%lims_r(1,i_rd),vac%lims_r(2,i_rd)
                        ! set local row index
                        rdl = sum(vac%lims_r(2,1:i_rd-1)-&
                            &vac%lims_r(1,1:i_rd-1)+1) + &
                            &rd-vac%lims_r(1,i_rd)+1

                        ! spherical radius
                        r_sph(rdl) = sqrt(sum(vac%x_vec(rd,:)**2))

                        ! test 1
                        phi(rdl,1) = vac%x_vec(rd,1)**vac%prim_X
                        dphi(rdl,1) = - vac%prim_X*vac%norm(rd,1)/&
                            &vac%x_vec(rd,1) * phi(rdl,1)

                        ! test 2
                        phi(rdl,2) = (vac%x_vec(rd,1)/&
                            &(r_sph(rdl)+abs(vac%x_vec(rd,2))))**vac%prim_X
                        dphi(rdl,2) = vac%prim_X*(vac%norm(rd,2)-&
                            &vac%norm(rd,1)*vac%x_vec(rd,2)/&
                            &vac%x_vec(rd,1)) / &
                            &r_sph(rdl)*sign(1._dp,vac%x_vec(rd,2)) * &
                            &phi(rdl,2)
                    end do row
                end do subrows
            else
                allocate(res(0,3,2))
                allocate(r_sph(0))
                allocate(phi(0,2))
                allocate(dphi(0,2))
            end if

            ! output
            allocate(loc_ser(vac%n_bnd,3))
            ierr = vec_dis2loc(vac%ctxt_HG,r_sph,vac%lims_r,loc_ser(:,1),&
                &proc=n_procs-1)
            chckerr('')
            if (rank.eq.n_procs-1) then
                plot_title(1) = 'spherical r'
                plot_name = 'r_sph'
                call print_ex_2d(plot_title(1),plot_name,loc_ser(:,1),&
                    &x=vac%ang(:,1),draw=.false.)
                call draw_ex([plot_title],plot_name,1,1,.false.)
            end if
            do jd = 1,size(phi,2)
                ierr = vec_dis2loc(vac%ctxt_HG,phi(:,jd),vac%lims_r,&
                    &loc_ser(:,jd),proc=n_procs-1)
                chckerr('')
            end do
            if (rank.eq.n_procs-1) then
                plot_title(1) = 'test potential '
                plot_name = 'phi'
                call print_ex_2d([plot_title(1)],plot_name,&
                    &loc_ser(:,1:size(phi,2)),x=vac%ang(:,1:1),draw=.false.)
                call draw_ex([plot_title(1)],plot_name,size(phi,2),1,&
                    &.false.)
            end if
            do jd = 1,size(dphi,2)
                ierr = vec_dis2loc(vac%ctxt_HG,dphi(:,jd),vac%lims_r,&
                    &loc_ser(:,jd),proc=n_procs-1)
                chckerr('')
            end do
            if (rank.eq.n_procs-1) then
                plot_title(1) = 'normal derivative of test potential d/dn'
                plot_name = 'dphi'
                call print_ex_2d([plot_title(1)],plot_name,&
                    &loc_ser(:,1:size(dphi,2)),x=vac%ang(:,1:1),&
                    &draw=.false.)
                call draw_ex([plot_title(1)],plot_name,size(dphi,2),1,&
                    &.false.)
            end if
            deallocate(loc_ser)

            ! loop over test potentials
            do jd = 1,size(phi,2)
                ! calculate H phi
                call pdgemv('N',vac%n_bnd,vac%n_bnd,1._dp,vac%H,1,1,&
                    &vac%desc_H,phi(:,jd),1,1,desc_res,1,0._dp,res(:,1,jd),&
```

```
                        &1,1,desc_res,1)

                    ! calculate - G dphi
                    call pdgemv('N',vac%n_bnd,vac%n_bnd,-1._dp,vac%G,1,1,&
                        &vac%desc_G,dphi(:,jd),1,1,desc_res,1,0._dp,&
                        &res(:,2,jd),1,1,desc_res,1)

                    ! add them together
                    res(:,3,jd) = sum(res(:,1:2,jd),2)
                end do

                ! output of tests
                allocate(loc_ser(vac%n_bnd,3))
                do id = 1,size(phi,2)
                    do jd = 1,3
                        ierr = vec_dis2loc(vac%ctxt_HG,res(:,jd,id),vac%lims_r,&
                            &loc_ser(:,jd),proc=n_procs-1)
                        chckerr('')
                    end do
                    if (rank.eq.n_procs-1) then
                        select case (id)
                            case (1)
                                plot_title(1) = 'H R^n'
                                plot_title(2) = '-G n Z_θ R^n'
                                plot_title(3) = '(H - G n Z_θ) R^n'
                                plot_name = 'test_vac_1'
                            case (2)
                                plot_title(1) = 'H (R/(r+|Z|))^n'
                                plot_title(2) = &
                                    &'- G |Z|/Z n dr/dθ (R/(r+|Z|))^n'
                                plot_title(3) = '(H - G |Z|/Z n dr/dθ) &
                                    &(R/(r+|Z|))^n'
                                plot_name = 'test_vac_2'
                        end select
                        call print_ex_2d(plot_title,trim(plot_name)//'_all',&
                            &loc_ser(:,:),x=vac%ang(:,1:1),&
                            &draw=.false.)
                        call draw_ex(plot_title,trim(plot_name)//'_all',3,1,&
                            &.false.)
                        call print_ex_2d(plot_title(3),plot_name,&
                            &loc_ser(:,3),x=vac%ang(:,1),&
                            &draw=.false.)
                        call draw_ex(plot_title(3:3),plot_name,1,1,.false.)
                    end if
                end do
                deallocate(loc_ser)
            end if
        end if
#endif

        call lvl_ud(-1)
    end function calc_gh

    !> \public Calculates matrices \c G and \c H in 3-D configuration.
    !!
    !! \see calc_gh.
    integer function calc_gh_1(vac,n_r_in,lims_r_in,x_vec_in,G_in,H_in,ext_in) &
        &result(ierr)

        use vac_utilities, only: calc_gh_int_1
        use grid_vars, only: min_par_x, max_par_x, min_alpha, max_alpha, n_alpha
        use rich_vars, only: n_par_x
        use num_vars, only: tol_zero
        use vac_vars, only: in_context
        use vac_utilities, only: vec_dis2loc

        character(*), parameter :: rout_name = 'calc_GH_1'

        ! input / output
        type(vac_type), intent(inout) :: vac                                    !< vacuum variables
        integer, intent(in) :: n_r_in                                           !< total number of rows of external poins of influence
        integer, intent(in) :: lims_r_in(:,:)                                   !< row limits of external points of influence
        real(dp), intent(in) :: x_vec_in(:,:)                                   !< position of influence
        real(dp), intent(in), pointer :: g_in(:,:)                              !< G at position of influence
        real(dp), intent(in), pointer :: h_in(:,:)                              !< H at position of influence
        logical, intent(in) :: ext_in                                           !< position of influence is external

        ! local variables
        integer :: kd                                                           ! counter
        integer :: i_rd, i_cd                                                   ! index of subrow and column
        integer :: rd, cd                                                       ! global counters for row and column
        integer :: rdl, cdl                                                     ! local counters for row and column, used for G and H
        integer :: lims_c_loc(2)                                                ! local column limits
        integer :: desc_res(blacsctxtsize)                                      ! descriptor for the result
```

```
        real(dp) :: dpar_x                                              ! step size in par_X
        real(dp) :: dalpha                                              ! step size in alpha
        real(dp) :: tol_loc                                             ! local tolerance on rho for singular points
        real(dp) :: g_loc(2)                                            ! local G
        real(dp) :: h_loc(2)                                            ! local H
        real(dp), allocatable :: res(:,:)                               ! sums of rows of H and vector of ones
        real(dp), allocatable :: loc_res(:)                             ! local res

        ! initialize ierr
        ierr = 0

        call writo('Using field-line 3-D Boundary Element Method')

        ! initialize variables
        dpar_x = (max_par_x-min_par_x)/(n_par_x-1)*pi
        if (n_alpha.gt.1) then
            dalpha = (max_alpha-min_alpha)/(n_alpha-1)*pi
        else
            dalpha = 0._dp
        end if

        ! set up local tolerance
        tol_loc = tol_zero*10                                           ! tolerance for singular interval

        ! iterate over all subrows: position of singular point
        subrows: do i_rd = 1,size(lims_r_in,2)
            ! iterate over all rows of this subrow
            row: do rd = lims_r_in(1,i_rd),lims_r_in(2,i_rd)
                ! set local row index
                rdl = sum(lims_r_in(2,1:i_rd-1)-lims_r_in(1,1:i_rd-1)+1) + &
                    &rd-lims_r_in(1,i_rd)+1

                ! iterate over all subcolumns: subintegrals
                subcols: do i_cd = 1,size(vac%lims_c,2)
                    ! set up local limits, including possible overlap
                    lims_c_loc(1) = max(1,vac%lims_c(1,i_cd)-1)
                    lims_c_loc(2) = min(vac%n_bnd,vac%lims_c(2,i_cd)+1)

                    ! iterate over all pairs of columns of this subcolumn
                    ! (upper limit is one lower as lims_c_loc
                    col: do cd = lims_c_loc(1),lims_c_loc(2)-1
                        ! set local column index
                        cdl = sum(vac%lims_c(2,1:i_cd-1)-&
                            &vac%lims_c(1,1:i_cd-1)+1) + &
                            &cd-vac%lims_c(1,i_cd)+1

                        ! calculate contributions to H and G
                        call calc_gh_int_1(g_loc,h_loc,&
                            &vac%x_vec(cd:cd+1,:),x_vec_in(rd,:),&
                            &vac%norm(cd:cd+1,:),vac%h_fac(rd,:),&
                            &[dpar_x,dalpha],tol_loc**2)

                        ! loop over left and right side of interval
                        do kd = 0,1
                            if (cd+kd.ge.vac%lims_c(1,i_cd) .and. &     ! lower local bound
                                &cd+kd.le.vac%lims_c(2,i_cd)) then      ! upper local bound
                                ! cycle if first or last global point
                                if (.not.on_field_line(cd,kd,vac%n_ang(1))) &
                                    &cycle
                                g_in(rdl,cdl+kd) = g_in(rdl,cdl+kd) + &
                                    &g_loc(1+kd)
                                h_in(rdl,cdl+kd) = h_in(rdl,cdl+kd) + &
                                    &h_loc(1+kd)
                            end if
                        end do
                    end do col
                end do subcols
            end do row
        end do subrows

        ! if in context  and not external, indirctly  calculate contribution due
        ! to fundament solution through constant potential
        if (in_context(vac%ctxt_HG) .and. .not.ext_in) then
            if (vac%lims_c(1,1).eq.1) then                              ! this process owns (part of) first column
                allocate(res(vac%n_loc(1),2))
            else
                allocate(res(0,2))
            end if
            call descinit(desc_res,vac%n_bnd,1,vac%bs,vac%bs,0,0,&
                &vac%ctxt_HG,max(1,vac%n_loc(1)),ierr)
            chckerr('descinit failed for res')

            ! set diagonal of H to zero
            ! iterate over all subrows: position of singular point
            subrows2: do i_rd = 1,size(lims_r_in,2)
                ! iterate over all rows of this subrow
                row2: do rd = lims_r_in(1,i_rd),lims_r_in(2,i_rd)
```

```
                    ! set local row index
                    rdl = sum(lims_r_in(2,1:i_rd-1)-&
                        &lims_r_in(1,1:i_rd-1)+1) + &
                        &rd-lims_r_in(1,i_rd)+1

                    ! iterate over all subcolumns: subintegrals
                    subcols2: do i_cd = 1,size(vac%lims_c,2)
                        cd = rd
                        if (vac%lims_c(1,i_cd).le.cd .and. &
                            &cd.le.vac%lims_c(2,i_cd)) then                    ! this subcolumn includes the diagonal
                            ! set local column index
                            cdl = sum(vac%lims_c(2,1:i_cd-1)-&
                                &vac%lims_c(1,1:i_cd-1)+1) + &
                                &cd-vac%lims_c(1,i_cd)+1

                            ! set diagonal of H to zero
                            h_in(rdl,cdl) = 0._dp
                        end if
                    end do subcols2
                end do row2
            end do subrows2

            ! set res(1) to ones
            res(:,1) = 1._dp

            ! calculate sum of rows of H
            call pdgemv('N',vac%n_bnd,vac%n_bnd,1._dp,vac%H,1,1,&
                &vac%desc_H,res(:,1),1,1,desc_res,1,0._dp,res(:,2),&
                &1,1,desc_res,1)

            ! add 4pi
            res(:,2) = res(:,2) + 4*pi

            ! distribute to all processes
            allocate(loc_res(vac%n_bnd))
            ierr = vec_dis2loc(vac%ctxt_HG,res(:,2),vac%lims_r,loc_res,&
                &proc=-1)
            chckerr('')

            ! Put in diagonal of H
            ! iterate over all subrows: position of singular point
            subrows3: do i_rd = 1,size(lims_r_in,2)
                ! iterate over all rows of this subrow
                row3: do rd = lims_r_in(1,i_rd),lims_r_in(2,i_rd)
                    ! set local row index
                    rdl = sum(lims_r_in(2,1:i_rd-1)-&
                        &lims_r_in(1,1:i_rd-1)+1) + &
                        &rd-lims_r_in(1,i_rd)+1

                    ! iterate over all subcolumns: subintegrals
                    subcols3: do i_cd = 1,size(vac%lims_c,2)
                        cd = rd
                        if (vac%lims_c(1,i_cd).le.cd .and. &
                            &cd.le.vac%lims_c(2,i_cd)) then                    ! this subcolumn includes the diagonal
                            ! set local column index
                            cdl = sum(vac%lims_c(2,1:i_cd-1)-&
                                &vac%lims_c(1,1:i_cd-1)+1) + &
                                &cd-vac%lims_c(1,i_cd)+1

                            ! set diagonal of H to inverse of sum
                            h_in(rdl,cdl) = - loc_res(rd)
                        end if
                    end do subcols3
                end do row3
            end do subrows3

            ! clean up
            deallocate(res)
            deallocate(loc_res)
        end if
    contains
        ! checks whether an interval is on the field line
        logical function on_field_line(cd,kd,n_ang) result(res)
            ! input / output
            integer, intent(in) :: cd                                          ! position of interval on field line
            integer, intent(in) :: kd                                          ! left or right point of interval
            integer, intent(in) :: n_ang                                       ! number of points on field line

            res = .true.
            if (mod(cd,n_ang).eq.1 .and. kd.eq.0) res = .false.                ! start of field line
            if (mod(cd,n_ang).eq.0 .and. kd.eq.1) res = .false.                ! end of field line
        end function on_field_line
    end function calc_gh_1

    !> \public Calculates matrices \c G and \c H in axisymmetric configurations.
    !!
    !! It makes use of toroidal functions.
```

```
    !!
    !! \see calc_gh.
    integer function calc_gh_2(vac,n_r_in,lims_r_in,x_vec_in,G_in,H_in,ext_in) &
        &result(ierr)

        use dtorh, only: dtorh1
        use vac_utilities, only: calc_gh_int_2
        use mpi_utilities, only: get_ser_var
        use num_ops, only: calc_zero_hh, calc_zero_zhang
        use num_vars, only: rank

        character(*), parameter :: rout_name = 'calc_GH_2'

        ! input / output
        type(vac_type), intent(inout) :: vac                                    !< vacuum variables
        integer, intent(in) :: n_r_in                                           !< total number of rows of external poins of influence
        integer, intent(in) :: lims_r_in(:,:)                                   !< row limits of external points of influence
        real(dp), intent(in) :: x_vec_in(:,:)                                   !< position of influence
        real(dp), intent(in), pointer :: g_in(:,:)                              !< G at position of influence
        real(dp), intent(in), pointer :: h_in(:,:)                              !< H at position of influence
        logical, intent(in) :: ext_in                                           !< position of influence is external

        ! local variables
        integer :: kd                                                           ! counter
        integer :: i_rd, i_cd                                                   ! index of subrow and column
        integer :: rd, cd                                                       ! global counters for row and column
        integer :: rdl, cdl                                                     ! local counters for row and column, used for G and H
        integer :: max_n                                                        ! maximum degree
        integer :: lims_c_loc(2)                                                ! local column limits
        real(dp) :: b_coeff(2)                                                  ! sum_k=1^n 2/2k-1 for n and n-1
        real(dp) :: tol_loc                                                     ! local tolerance on rho for singular points
        real(dp) :: del_r(2,2)                                                  ! unit vectors of next and previous subinterval
        real(dp) :: g_loc(2)                                                    ! local G
        real(dp) :: h_loc(2)                                                    ! local H
        real(dp) :: beta_comp                                                   ! complementary beta
        real(dp) :: ang                                                         ! local angle
        real(dp), parameter :: tol_q = 1.e-6_dp                                 ! tolerance on Q approximation
        real(dp), allocatable :: beta(:)                                        ! beta
        real(dp), allocatable :: loc_ser(:)                                     ! dummy serial variable
        real(dp), allocatable :: rho2(:)                                        ! square of distance in projected poloidal plane for one column
        real(dp), allocatable :: arg(:)                                         ! argument 1 + rho^2/(RiRj)(
        real(dp), allocatable :: aij(:)                                         ! Aij helper variable for H
        real(dp), allocatable :: ql(:,:,:)                                      ! ql for all rho2's
        real(dp), allocatable :: pl_loc(:), ql_loc(:)                           ! local toroidal functions
        logical, allocatable :: was_calc(:,:)                                   ! was already calculated
        logical, allocatable :: sing_col(:)                                     ! singular column, used only with external influence
        logical :: was_calc_sym                                                 ! symmetrical was_calc
        character(len=max_str_ln) :: err_msg                                    ! error message
#if ldebug
        real(dp) :: rho2_lims(2)                                                ! limits on rho2
        real(dp) :: gamma_lims(2)                                               ! limits on gamma
        integer :: beta_lims(2)                                                 ! limits to plot beta
        !integer :: lims_cl(2)                                                   ! vac%lims_c in local coordinates
        !character(len=max_str_ln) :: plot_title                                ! plot title
        !character(len=max_str_ln) :: plot_name                                 ! plot name
#endif

        ! initialize ierr
        ierr = 0

        call writo('Using axisymmetric Boundary Element Method')

        ! set up b_coeff and local tolerance
        b_coeff = 0._dp
        do kd = 1,vac%prim_X
            b_coeff(2) = b_coeff(2) + 2._dp/(2*kd-1)
        end do
        b_coeff(1) = b_coeff(2) - 2._dp/(2*vac%prim_X-1)
        err_msg = calc_zero_zhang(tol_loc,fun_tol,&
            &[1.e-10_dp,&                                                       ! very low lower limit
            &exp(-2*b_coeff(2))])                                               ! upper limit for -1/2ln(x) = b
        if (trim(err_msg).ne.'') then
            ierr = 1
            chckerr(trim(err_msg))
        end if
        tol_loc = 8*minval(vac%x_vec(:,1))*tol_loc                              ! tolerance on rho
        ierr = get_ser_var([tol_loc],loc_ser,scatter=.true.)
        chckerr('')
        tol_loc = maxval(loc_ser)
        deallocate(loc_ser)
        call writo('Tolerance on rho for analytical integral: '//&
```

```
            &trim(r2str(tol_loc)))

        ! allocate helper variables
        allocate(was_calc(1:n_r_in,1:vac%n_bnd))
        was_calc = .false.                                          ! will remain false if external influence
        allocate(ql(1:n_r_in,1:vac%n_bnd,2))
        allocate(pl_loc(0:max(vac%prim_X,1)))
        allocate(ql_loc(0:max(vac%prim_X,1)))
#if ldebug
        rho2_lims = [huge(1._dp),-huge(1._dp)]
        gamma_lims = [huge(1._dp),-huge(1._dp)]
#endif

        ! iterate over all subrows: position of singular point
        subrows: do i_rd = 1,size(lims_r_in,2)
            ! initialize helper variables
            allocate(beta(lims_r_in(1,i_rd):lims_r_in(2,i_rd)))     ! global index
#if ldebug
            beta_lims = [lims_r_in(2,i_rd)+1,lims_r_in(1,i_rd)-1]
#endif

            ! iterate over all rows of this subrow
            row: do rd = lims_r_in(1,i_rd),lims_r_in(2,i_rd)
                ! set local row index
                rdl = sum(lims_r_in(2,1:i_rd-1)-lims_r_in(1,1:i_rd-1)+1) + &
                    &rd-lims_r_in(1,i_rd)+1

                ! iterate over all subcolumns: subintegrals
                subcols: do i_cd = 1,size(vac%lims_c,2)
                    ! set up local limits, including possible overlap
                    lims_c_loc(1) = max(1,vac%lims_c(1,i_cd)-1)
                    lims_c_loc(2) = min(vac%n_bnd,vac%lims_c(2,i_cd)+1)

                    ! initialize helper variables
                    allocate(rho2(lims_c_loc(1):lims_c_loc(2)))     ! global index
                    allocate(arg(lims_c_loc(1):lims_c_loc(2)))      ! global index
                    allocate(aij(lims_c_loc(1):lims_c_loc(2)))      ! global index

                    ! initialize singular column flags
                    if (ext_in) then
                        if (allocated(sing_col)) deallocate(sing_col)
                        allocate(sing_col(&
                            &vac%lims_c(1,i_cd):vac%lims_c(2,i_cd)))
                        sing_col = .false.
                    end if

                    ! iterate over all columns  of this subcolumn, possibly with
                    ! an overlap left and/or right to set up the Aij
                    col: do cd = lims_c_loc(1),lims_c_loc(2)
                        ! check rho2 and set arg to calculate tor. functions
                        rho2(cd) = sum((vac%x_vec(cd,:)-x_vec_in(rd,:))**2)
                        arg(cd) = 1._dp + &
                            &rho2(cd)/(2*vac%x_vec(cd,1)*x_vec_in(rd,1))
                        was_calc_sym = .false.
                        if (.not.ext_in) was_calc_sym = was_calc(cd,rd)
                        if (rho2(cd).le.tol_loc**2) then
                            ! don't set it
                        else if (was_calc_sym) then                 ! check symmetric element
                            ql(rd,cd,:) = ql(cd,rd,:)               ! set it
                        else                                        ! not yet calculated by this process
#if ldebug
                            ! update extrema
                            if (rho2(cd).lt.rho2_lims(1)) &
                                &rho2_lims(1) = rho2(cd)
                            if (rho2(cd).gt.rho2_lims(2)) &
                                &rho2_lims(2) = rho2(cd)
                            if (arg(cd).lt.gamma_lims(1)) &
                                &gamma_lims(1) = arg(cd)
                            if (arg(cd).gt.gamma_lims(2)) &
                                &gamma_lims(2) = arg(cd)
#endif
                            err_msg = 'The solutions did not converge for &
                                &rho^2 = '//trim(r2str(rho2(cd)))//&
                                &' and n = '//trim(i2str(vac%prim_X))
                            if (vac%prim_X.gt.0) then               ! positive n
                                ierr = dtorh1(arg(cd),0,vac%prim_X,pl_loc,&
                                    &ql_loc,max_n)
                                chckerr('')
                                if (max_n.lt.vac%prim_X) then
                                    ierr = 1
                                    chckerr(err_msg)
                                end if
                                ql(rd,cd,:) = ql_loc(vac%prim_X-1:vac%prim_X)
                            else if (vac%prim_X.eq.0) then
                                ierr = dtorh1(arg(cd),0,1,pl_loc,ql_loc,max_n)
                                chckerr('')
                                if (max_n.lt.vac%prim_X) then
```

```
                                ierr = 1
                                chckerr(err_msg)
                            end if
                            ql(rd,cd,1) = ql_loc(1)                         ! Q_-3/2 = Q_1/2
                            ql(rd,cd,2) = ql_loc(0)                         ! Q_-1/2
                        else                                                ! negative n
                            ierr = 1
                            err_msg = 'negative n not yet implemented'      ! SHOULD JUST BE Q_{n-1/2} = Q_{-n-1/2}
                            chckerr('')
                        end if
                        if (.not.ext_in) was_calc(rd,cd) = .true.
                    end if

                    ! set up Aij
                    if (rho2(cd).le.tol_loc**2) then
                        if (ext_in) then
                            if (cd.ge.vac%lims_c(1,i_cd) .and. &
                                &cd.le.vac%lims_c(2,i_cd)) &
                                &sing_col(cd) = .true.
                            !!Aij(cd) = 0._dp
                        else
                            aij(cd) = 0.5_dp*(vac%prim_X - 0.5_dp) * &
                                &(x_vec_in(rd,1) * &
                                &(vac%norm(rd,1)*vac%dnorm(rd,2)-&
                                &vac%norm(rd,2)*vac%dnorm(rd,1))/&
                                &sum(vac%norm(rd,:)**2) &
                                &+vac%norm(rd,1)/x_vec_in(rd,1))
                        end if
                    else
                        aij(cd) = 2._dp*x_vec_in(rd,1)*vac%x_vec(cd,1) / &
                            &(4._dp*x_vec_in(rd,1)*vac%x_vec(cd,1)+&
                            &rho2(cd)) *(vac%prim_X - 0.5_dp) * &
                            &(- 2._dp/rho2(cd) * sum(vac%norm(cd,:)*&
                            &(vac%x_vec(cd,:)-x_vec_in(rd,:))) + &
                            &vac%norm(cd,1)/vac%x_vec(cd,1))
                    end if
                end do col

#if ldebug
                if (debug_calc_gh) then
                    !! output
                    !plot_title = 'for row '//trim(i2str(rd))//&
                        !&' and starting column '//trim(i2str(lims_c_loc(1)))
                    !plot_name = '_'//trim(i2str(rd))//'_'//&
                        !&trim(i2str(lims_c_loc(1)))
                    !call print_ex_2D('rho^2 '//trim(plot_title),&
                        !&'rho2'//trim(plot_name),rho2,&
                        !&x=vac%ang(lims_c_loc(1):lims_c_loc(2),1),&
                        !&draw=.false.)
                    !call draw_ex(['rho^2 '//trim(plot_title)],&
                        !&'rho2'//trim(plot_name),1,1,.false.)
                    !call print_ex_2D('γ '//trim(plot_title),&
                        !&'gamma'//trim(plot_name),arg,&
                        !&x=vac%ang(lims_c_loc(1):lims_c_loc(2),1),&
                        !&draw=.false.)
                    !call draw_ex(['γ '//trim(plot_title)],&
                        !&'gamma'//trim(plot_name),1,1,.false.)
                    !call print_ex_2D('A_{ij} '//trim(plot_title),&
                        !&'Aij'//trim(plot_name),arg,&
                        !&x=vac%ang(lims_c_loc(1):lims_c_loc(2),1),&
                        !&draw=.false.)
                    !call draw_ex(['A_{ij} '//trim(plot_title)],&
                        !&'Aij'//trim(plot_name),1,1,.false.)
                end if
#endif

                ! iterate over all pairs of columns of this subcolumn
                ! (upper limit is one lower as previous)
                col2: do cd = lims_c_loc(1),lims_c_loc(2)-1
                    ! set local column index
                    cdl = sum(vac%lims_c(2,1:i_cd-1)-&
                        &vac%lims_c(1,1:i_cd-1)+1) + &
                        &cd-vac%lims_c(1,i_cd)+1

                    ! check for singular columns if external influence
                    if (ext_in) then
                        if (any(sing_col)) then
                            do kd = 0,1
                                if (cd+kd.ge.vac%lims_c(1,i_cd) .and. &
                                    &cd+kd.le.vac%lims_c(2,i_cd)) then
                                    g_in(rdl,cdl+kd) = 0._dp
                                    h_in(rdl,cdl+kd) = 4*pi
                                end if
                            end do
                            cycle col2                                      ! pass to next column
                        end if
                    end if
```

```

                        ! set ang
                        if (ext_in) then
                            ang = 0._dp
                        else
                            ang = vac%ang(rd,1)
                        end if

                        ! calculate contributions to H and G
                        call calc_gh_int_2(g_loc,h_loc,&
                            &vac%ang(cd:cd+1,1),ang,&
                            &vac%x_vec(cd:cd+1,:),x_vec_in(rd,:),&
                            &vac%norm(cd:cd+1,:),vac%norm(rd,:),&
                            &aij(cd:cd+1),ql(rd,cd:cd+1,:),&
                            &tol_loc**2,b_coeff,vac%prim_X)

                        do kd = 0,1
                            if (cd+kd.ge.vac%lims_c(1,i_cd) .and. &
                                &cd+kd.le.vac%lims_c(2,i_cd)) then
                                g_in(rdl,cdl+kd) = g_in(rdl,cdl+kd) + &
                                    &g_loc(1+kd)
                                h_in(rdl,cdl+kd) = h_in(rdl,cdl+kd) + &
                                    &h_loc(1+kd)
                            end if
                        end do
                    end do col2

#if ldebug
                    if (debug_calc_gh) then
                        !! local limits
                        !lims_cl = sum(vac%lims_c(2,1:i_cd-1)-&
                            !&vac%lims_c(1,1:i_cd-1)+1) + &
                            !&vac%lims_c(:,i_cd)-vac%lims_c(1,i_cd)+1
                        !! output
                        !call print_ex_2D('G '//trim(plot_title),&
                            !&'G'//trim(plot_name),&
                            !&G_in(rdl,lims_cl(1):lims_cl(2)),x&
                            !&=vac%ang(vac%lims_c(1,i_cd):vac%lims_c(2,i_cd),1),&
                            !&draw=.false.)
                        !call draw_ex(['G '//trim(plot_title)],&
                            !&'G'//trim(plot_name),1,1,.false.)
                        !call print_ex_2D('H '//trim(plot_title),&
                            !&'H'//trim(plot_name),&
                            !&H_in(rdl,lims_cl(1):lims_cl(2)),x&
                            !&=vac%ang(vac%lims_c(1,i_cd):vac%lims_c(2,i_cd),1),&
                            !&draw=.false.)
                        !call draw_ex(['H '//trim(plot_title)],&
                            !&'H'//trim(plot_name),1,1,.false.)
                    end if
#endif

                    ! calculate contribution due to fundamental solution
                    ! (only if not external)
                    if (.not.ext_in .and. vac%lims_c(1,i_cd).le.rd .and. &
                        &rd.le.vac%lims_c(2,i_cd)) then                        ! this subcolumn includes the diagonal
                        ! set local column index
                        cdl = sum(vac%lims_c(2,1:i_cd-1)-&
                            &vac%lims_c(1,1:i_cd-1)+1) + &
                            &rd-vac%lims_c(1,i_cd)+1

                        if (rd.eq.1 .or. rd.eq.vac%n_bnd) then                 ! first or last global point
                            ! loop around to get previous direction vector
                            del_r(1,:) = vac%x_vec(vac%n_bnd,:)-&
                                &vac%x_vec(vac%n_bnd-1,:)
                            del_r(2,:) = vac%x_vec(2,:)-vac%x_vec(1,:)
                        else                                                   ! all points in between
                            del_r(1,:) = vac%x_vec(rd,:)-vac%x_vec(rd-1,:)
                            del_r(2,:) = vac%x_vec(rd+1,:)-vac%x_vec(rd,:)
                        end if
                        do kd = 1,2
                            del_r(kd,:) = del_r(kd,:)/sqrt(sum(del_r(kd,:)**2))
                        end do
                        beta_comp = acos(sum(product(del_r,1)))
                        if (del_r(1,1)*del_r(2,2) .lt. &
                            &del_r(1,2)*del_r(2,1)) beta_comp = - beta_comp     ! concave
                        beta(rd) = beta_comp + pi
#if ldebug
                        if (debug_calc_gh) then
                            if (rd.lt.beta_lims(1)) beta_lims(1) = rd
                            if (rd.gt.beta_lims(2)) beta_lims(2) = rd
                        end if
#endif

                        h_in(rdl,cdl) = h_in(rdl,cdl) - 2*beta(rd)
                    end if

                    ! clean up
```

```
                    deallocate(arg,rho2,aij)
                end do subcols
            end do row

#if ldebug
            if (debug_calc_gh) then
                !! output
                !if (beta_lims(2).ge.beta_lims(1)) then
                    !plot_title = 'beta/n  for starting row '//&
                        !&trim(i2str(beta_lims(1)))
                    !plot_name = 'beta_'//trim(i2str(beta_lims(1)))
                    !call print_ex_2D(plot_title,plot_name,&
                        !&beta(beta_lims(1):beta_lims(2))/pi,&
                        !&x=vac%ang(beta_lims(1):beta_lims(2),1),draw=.false.)
                    !call draw_ex([plot_title],plot_name,1,1,.false.)
                !end if
            end if
#endif

            ! clean up
            deallocate(beta)
        end do subrows

#if ldebug
        if (debug_calc_gh) then
            ! output
            ierr = get_ser_var([rho2_lims(1)],loc_ser)
            chckerr('')
            if (rank.eq.0) rho2_lims(1) = minval(loc_ser)
            deallocate(loc_ser)
            ierr = get_ser_var([rho2_lims(2)],loc_ser)
            chckerr('')
            if (rank.eq.0) rho2_lims(2) = maxval(loc_ser)
            deallocate(loc_ser)
            ierr = get_ser_var([gamma_lims(1)],loc_ser)
            chckerr('')
            if (rank.eq.0) gamma_lims(1) = minval(loc_ser)
            deallocate(loc_ser)
            ierr = get_ser_var([gamma_lims(2)],loc_ser)
            chckerr('')
            if (rank.eq.0) gamma_lims(2) = maxval(loc_ser)
            deallocate(loc_ser)
            if (rank.eq.0) then
                call writo('limits in rho: '//trim(r2str(sqrt(rho2_lims(1))))//&
                    &'..'//trim(r2str(sqrt(rho2_lims(2)))))
                call writo('limits in gamma: '//trim(r2str(gamma_lims(1)))//&
                    &'..'//trim(r2str(gamma_lims(2))))
            end if
        end if
#endif
    contains
        ! function that returns f = -1/2 ln(x)-b - x/T.
        !
        ! It uses b_coeff (b) and tol_Q (T)  from the parent function where T is
        ! a hard-coded positive value << 1.
        !
        !> \private
        function fun_tol(val)
            ! input / output
            real(dp), intent(in) :: val
            real(dp) :: fun_tol

            fun_tol = -0.5_dp*log(val)-b_coeff(2)-val/tol_q
        end function fun_tol
    end function calc_gh_2

    !> Calculates the vacuum response.
    !!
    !! First  \c G and \c  H are completed if  this is not the  first Richardson
    !! level.
    !!
    !! Then the vacuum contribution is calculated using the two-fold strategy of
    !! first solving
    !!  \f$\overline{\text{H}}\overline{\Phi} =
    !!  \overline{\text{G}}\overline{\text{E}} \overline{\text{P}}\f$,
    !! followed by left-multiplication of \f$\overline{\Phi}\f$ by
    !!  \f$\overline{\text{P}}\overline{\text{E}}^\dagger\f$.
    !!
    !! The  non-square matrix  \f$\overline{\text{E}}\f$ contains  the exponents
    !! for different  mode numbers and  different poloidal grid  points, whereas
    !! \f$\overline{\text{P}}\f$ are diagonal matrices  that contain the factors
    !! \f$(nq-m)\f$.
    !!
    !! In practice, the complex matrix \f$\overline{\text{E}}\f$ is split in the
    !! two components of a real matrix twice the width.
    !!
    !! For vacuum style  1, the poloidal grid points correspond  to the paralell
```

```
    !! grid points and have to be provied by a \c grid variable.
    !!
    !! If \c  jump_to_sol is used for  the current Richardson level,  the vacuum
    !! quantities are not calculated, but just restored.
    !!
    !! Currently, this procedure only works for vacuum style 2 (axisymmetric).
    !!
    !! \return ierr
    integer function calc_vac_res(mds,vac) result(ierr)
        use grid_vars, only: min_par_x, max_par_x, min_alpha, max_alpha, n_alpha
        use mpi
        use rich_vars, only: rich_lvl, n_par_x
        use num_vars, only: jump_to_sol, rich_restart_lvl, eq_style, &
            &use_pol_flux_f, rank, n_procs
        use pb3d_ops, only: reconstruct_pb3d_vac
        use x_vars, only: n_mod_x
        use vac_vars, only: set_loc_lims, in_context
        use vac_utilities, only: mat_dis2loc
        use eq_vars, only: vac_perm
        use num_utilities, only: lcm
        use mpi_utilities, only: broadcast_var

        character(*), parameter :: rout_name = 'calc_vac_res'

        ! input / output
        type(modes_type), intent(in) :: mds                                     !< general modes variables
        type(vac_type), intent(inout) :: vac                                    !< vacuum variables

        ! local variables
        integer :: jd                                                           ! counter
        integer :: rd, cd                                                       ! global counter for row and col
        integer :: rdl, cdl                                                     ! local counter for row and col
        integer :: i_rd, i_cd                                                   ! index of subrow and subcol
        integer :: rich_lvl_name                                                ! either the Richardson level or zero, to
    append to names
        integer :: sec_x_loc                                                    ! local sec_X
        integer :: n_loc(2)                                                     ! n_loc for n_mod_X instead of n_bnd
        integer :: lims_cl(2)                                                   ! lims_c_loc in local coordinates
        integer :: par_id                                                       ! parallel index
        integer :: alpha_id                                                     ! field line label
        integer, allocatable :: lims_r_loc(:,:)                                 ! local row limits for n_loc
        integer, allocatable :: lims_c_loc(:,:)                                 ! local column limits for n_loc
        integer, target :: desc_phiep(blacsctxtsize)                           ! descriptor for Phi and EP
        integer, target :: desc_gep(blacsctxtsize)                             ! descriptor for GEP
        integer, target :: desc_res2(blacsctxtsize)                            ! descriptor for res2
        real(dp) :: arg_loc                                                     ! local argument
        real(dp) :: dpar_x                                                      ! step size in par_X
        real(dp) :: dalpha                                                      ! step size in alpha
        real(dp) :: i_loc                                                       ! local integration rule
        real(dp) :: par_loc                                                     ! local parallel angle
        real(dp) :: alpha_loc                                                   ! local field line label
        real(dp), allocatable, target :: ep(:,:)                                ! EP
        real(dp), allocatable, target :: gep(:,:)                               ! GEP
        real(dp), allocatable, target :: phi(:,:)                               ! intermediate Phi
        real(dp), allocatable, target :: res2(:,:)                              ! vacuum response, real version
        real(dp), allocatable, target :: res2_loc(:,:)                          ! local version of res2
        logical :: rank_o                                                       ! output rank
#if ldebug
        integer :: lwork                                                        ! size of work array
        real(dp), allocatable :: res2_ev(:,:)                                   ! eigenvalues of res2
        real(dp), allocatable :: tau(:)                                         ! array tau in upper Hessenberg form
        real(dp), allocatable :: work(:)                                        ! work array
        real(dp), allocatable :: x_plot(:,:)                                    ! X of plot
        real(dp), allocatable :: y_plot(:,:)                                    ! Y of plot
        complex(dp), allocatable :: res_loc(:,:)                                ! local vac%res
        complex(dp), allocatable :: res_ev(:)                                   ! eigenvalues of res2
        complex(dp), allocatable :: work_c(:)                                   ! work array
        complex(dp), allocatable :: tau_c(:)                                    ! array tau in upper Hessenberg form
#endif

        ! initialize ierr
        ierr = 0

        ! decide whether to append Richardson level to name
        select case (eq_style)
            case (1)                                                            ! VMEC
                rich_lvl_name = rich_lvl                                        ! append richardson level
            case (2)                                                            ! HELENA
                rich_lvl_name = 0                                               ! do not append name
        end select

        if (rich_lvl.eq.rich_restart_lvl .and. jump_to_sol) then
            ! reconstruct old vacuum
            ierr = reconstruct_pb3d_vac(vac,'vac',rich_lvl=rich_lvl_name)
            chckerr('')
            return
        end if
```

```fortran

        call writo('Start calculation of vacuum quantities')

        call lvl_ud(1)

        ! set ouptut rank persistency
        if (rank.eq.n_procs-1) then
            rank_o = .true.
        else
            rank_o = .false.
        end if

        ! calculate matrices G and H
        ierr = calc_gh(vac)
        chckerr('')

        ! Only for processes that are in the blacs context
        if (in_context(vac%ctxt_HG)) then
            ! user output
            call writo('Use STRUMPack to solve system',persistent=rank_o)
            call lvl_ud(1)

            ! set step sizes if 3-D vacuum
            if (vac%style.eq.1) then
                dpar_x = (max_par_x-min_par_x)/(n_par_x-1)*pi
                if (n_alpha.gt.1) then
                    dalpha = (max_alpha-min_alpha)/(n_alpha-1)*pi
                else
                    dalpha = 0._dp
                end if
            end if

            ! set secondary mode numbers
            allocate(vac%sec_X(n_mod_x))
            if (use_pol_flux_f) then
                vac%sec_X = mds%m(size(mds%m,1),:)
            else
                vac%sec_X = mds%n(size(mds%n,1),:)
            end if

            ! set sizes
            do jd = 1,2
                n_loc(jd) = numroc(2*n_mod_x,vac%bs,vac%ind_p(jd),0,vac%n_p(jd))
            end do
            call set_loc_lims(n_loc(1),vac%bs,vac%ind_p(1),vac%n_p(1),&
                &lims_r_loc)
            call set_loc_lims(n_loc(2),vac%bs,vac%ind_p(2),vac%n_p(2),&
                &lims_c_loc)

            ! initialize descriptors
            call descinit(desc_phiep,vac%n_bnd,2*n_mod_x,vac%bs,vac%bs,0,0,&
                &vac%ctxt_HG,max(1,vac%n_loc(1)),ierr)
            chckerr('descinit failed for EP')
            call descinit(desc_gep,vac%n_bnd,2*n_mod_x,vac%bs,vac%bs,0,0,&
                &vac%ctxt_HG,max(1,vac%n_loc(1)),ierr)
            chckerr('descinit failed for GEP')
            call descinit(desc_res2,2*n_mod_x,2*n_mod_x,vac%bs,vac%bs,0,0,&
                &vac%ctxt_HG,max(1,n_loc(1)),ierr)
            chckerr('descinit failed for res2')

            ! allocate auxililary matrices
            allocate(ep(vac%n_loc(1),n_loc(2)))
            allocate(gep(vac%n_loc(1),n_loc(2)))
            allocate(phi(vac%n_loc(1),n_loc(2)))
            allocate(res2(n_loc(1),n_loc(2)))
            allocate(res2_loc(2*n_mod_x,2*n_mod_x))

            ! set EP
            ! iterate over all subcolumns of matrix with 2n_mod_X columns
            subcols: do i_cd = 1,size(lims_c_loc,2)
                col: do cd = lims_c_loc(1,i_cd),lims_c_loc(2,i_cd)
                    ! set local column index
                    cdl = sum(lims_c_loc(2,1:i_cd-1)-&
                        &lims_c_loc(1,1:i_cd-1)+1) + &
                        &cd-lims_c_loc(1,i_cd)+1

                    ! set local secondary mode number
                    sec_x_loc = vac%sec_X(mod(cd-1,n_mod_x)+1)

                    ! iterate over all subrows of matrix with n_bnd rows
                    subrows: do i_rd = 1,size(vac%lims_r,2)
                        row: do rd = vac%lims_r(1,i_rd),vac%lims_r(2,i_rd)
                            ! set local row index
                            rdl = sum(vac%lims_r(2,1:i_rd-1)-&
                                &vac%lims_r(1,1:i_rd-1)+1) + &
                                &rd-vac%lims_r(1,i_rd)+1

```

```
                        ! set parallel index and field line label
                        par_id = mod(rd-1,vac%n_ang(1))+1
                        alpha_id = (rd-1)/vac%n_ang(1) + 1

                        ! set up local argument
                        select case (vac%style)
                            case (1)                                    ! field-line 3-D
                                par_loc = pi*min_par_x + &
                                    &(par_id-1)*dpar_x
                                alpha_loc = pi*min_alpha + &
                                    &(alpha_id-1)*dalpha
                                if (use_pol_flux_f) then
                                    arg_loc = vac%prim_X*alpha_loc + &
                                        &(vac%prim_X*vac%jq-sec_x_loc)*&
                                        &par_loc
                                else
                                    arg_loc = vac%prim_X*alpha_loc + &
                                        &(sec_x_loc-vac%prim_X*vac%jq)*&
                                        &par_loc
                                end if
                            case (2)                                    ! axisymmetric
                                arg_loc = -sec_x_loc*&
                                    &vac%ang(par_id,alpha_id)
                        end select

                        ! set up argument

                        ! save EP
                        if (cd.le.n_mod_x) then                         ! real part of rhs
                            ep(rdl,cdl) = cos(arg_loc)                  ! rp(i exp)
                        else
                            ep(rdl,cdl) = sin(arg_loc)                  ! ip(i exp)
                        end if
                        if (use_pol_flux_f) then
                            ep(rdl,cdl) = (vac%prim_X*vac%jq-sec_x_loc)*&
                                &ep(rdl,cdl)
                        else
                            ep(rdl,cdl) = (sec_x_loc-vac%prim_X*vac%jq)*&
                                &ep(rdl,cdl)
                        end if
                    end do row
                end do subrows
            end do col
        end do subcols
        call plot_hdf5('EP','EP',reshape(ep,[vac%n_bnd,n_loc(2),1]))

        ! solve for Phi
        ierr = solve_phi_bem(vac,ep,phi,[vac%n_bnd,2*n_mod_x],&
            &[vac%n_loc(1),n_loc(2)],lims_c_loc,desc_phiep)
        chckerr('')

        ! calculate GEP
        call pdgemm('N','N',vac%n_bnd,2*n_mod_x,vac%n_bnd,1._dp,vac%G,1,1,&
            &vac%desc_G,ep,1,1,desc_phiep,0._dp,gep,1,1,desc_gep)

        call lvl_ud(-1)

        ! user output
        call writo('Combine results into vacuum response',persistent=rank_o)
        call lvl_ud(1)

        ! Convert EP into IEP where I contains the integration rule:
        !         (θ_{i+1}-θ_{i-1})/2   for i = 2..n-1
        !   I_i = (θ_2-θ_1}/2           for i = 1
        !         (θ_{n}-θ_{n-1}}/2     for i = n
        ! with n = vac%n_bnd.
        ! For vacua of type 1, the step  size is constant. Also, there is an
        ! additional step size in the alpha direction.
        subrows3: do i_rd = 1,size(vac%lims_r,2)
            row3: do rd = vac%lims_r(1,i_rd),vac%lims_r(2,i_rd)
                ! set local row index
                rdl = sum(vac%lims_r(2,1:i_rd-1)-&
                    &vac%lims_r(1,1:i_rd-1)+1) + &
                    &rd-vac%lims_r(1,i_rd)+1

                subcols3: do i_cd = 1,size(lims_c_loc,2)
                    ! set local column limits
                    lims_cl = sum(lims_c_loc(2,1:i_cd-1)-&
                        &lims_c_loc(1,1:i_cd-1)+1) + &
                        &lims_c_loc(:,i_cd)-lims_c_loc(1,i_cd)+1

                    select case (vac%style)
                        case (1)                                        ! field-line 3-D
                            i_loc = dpar_x*dalpha
                            if ((rd-1)/vac%n_ang(1).eq.0 .or. &         ! first point on field line
                                &(rd-1)/vac%n_ang(1).eq.vac%n_ang(1)-1) &   ! last point on field line
                                &i_loc = i_loc*0.5_dp
```

```
                            case (2)                                        ! axisymmetric
                                i_loc = 0.5_dp*&
                                    &(vac%ang(min(rd+1,vac%n_bnd),1)-&
                                    &vac%ang(max(rd-1,1),1))
                        end select

                        ep(rdl,lims_cl(1):lims_cl(2)) = &
                            &ep(rdl,lims_cl(1):lims_cl(2)) * i_loc
                    end do subcols3
                end do row3
            end do subrows3

            ! calculate (EP)^T Phi
            ! Note that this means that the second  half of the rows in EP^T are
            ! still missing a  minus sign if they are to  represent EP^†. This
            ! will be taken into account when res2_loc is converted to vac%res.
            call pdgemm('T','N',2*n_mod_x,2*n_mod_x,vac%n_bnd,1._dp,ep,1,1,&
                &desc_phiep,phi,1,1,desc_phiep,0._dp,res2,1,1,desc_res2)

            ! get serial version on last process
            ierr = mat_dis2loc(vac%ctxt_HG,res2,lims_r_loc,lims_c_loc,res2_loc,&
                &proc=n_procs-1)
            chckerr('')

#if ldebug
            ! user output
            if (debug_calc_vac_res) then
                call writo('Calculate the eigenvalues of res_loc to &
                    &investigate whether it is positive definite',&
                    &persistent=rank_o)
                call lvl_ud(1)
            end if
#endif

            ! convert to complex numbers and save in vac%res
            if (rank.eq.n_procs-1) then
                vac%res = res2_loc(1:n_mod_x,1:n_mod_x)-&
                    &(-res2_loc(n_mod_x+1:2*n_mod_x,n_mod_x+1:2*n_mod_x))       ! minus sign for lower rows
                vac%res = vac%res + iu * &
                    &(res2_loc(1:n_mod_x,n_mod_x+1:2*n_mod_x)-&
                    &res2_loc(n_mod_x+1:2*n_mod_x,1:n_mod_x))                   ! minus sign for lower rows
                vac%res = vac%res/vac_perm
#if ldebug
                allocate(x_plot(n_mod_x,n_mod_x))
                allocate(y_plot(n_mod_x,n_mod_x))
                do jd = 1,n_mod_x
                    x_plot(jd,:) = vac%sec_X(jd)
                    y_plot(:, jd) = vac%sec_X(jd)
                end do
                call print_ex_2d(['real vacuum response'],'vac_res_Re',&
                    &rp(vac%res),x=x_plot,draw=.false.)
                call draw_ex(['real vacuum response'],'vac_res_Re',n_mod_x,1,&
                    &.false.)
                call print_ex_2d(['imag vacuum response'],'vac_res_Im',&
                    &ip(vac%res),x=x_plot,draw=.false.)
                call draw_ex(['imag vacuum response'],'vac_res_Im',n_mod_x,1,&
                    &.false.)
                call plot_hdf5(['real part','imag part','imag diff'],'vac_res',&
                    &reshape([rp(vac%res),ip(vac%res),&
                    &ip(vac%res+transpose(vac%res))],[n_mod_x,n_mod_x,1,3]),&
                    &x=reshape(x_plot,[n_mod_x,n_mod_x,1,1]),&
                    &y=reshape(y_plot,[n_mod_x,n_mod_x,1,1]))
                deallocate(x_plot,y_plot)

                if (debug_calc_vac_res) then
                    ! copy vacuum response
                    allocate(res_loc(n_mod_x,n_mod_x))
                    res_loc = vac%res

                    ! get size of work matrix required for zgehrd
                    allocate(work_c(1))
                    allocate(tau_c(n_mod_x-1))
                    call zgehrd(n_mod_x,1,n_mod_x,res_loc,n_mod_x,tau_c,work_c,&
                        &-1,ierr)
                    chckerr('zgehrd failed')
                    lwork = nint(rp(work_c(1)))
                    deallocate(work_c)
                    allocate(work_c(lwork))

                    ! convert to upper Heisenberg form
                    call zgehrd(n_mod_x,1,n_mod_x,res_loc,n_mod_x,tau_c,work_c,&
                        &lwork,ierr)
                    chckerr('zgehrd failed')
                    deallocate(work_c)

                    ! get size of work matrix required for zhseqr
                    allocate(work_c(1))
```

```
                    allocate(res_ev(n_mod_x))
                    call zhseqr('E','N',n_mod_x,1,n_mod_x,res_loc,n_mod_x,&
                        &res_ev,0._dp,n_mod_x,work_c,-1,ierr)
                    chckerr('zhseqr failed')
                    lwork = nint(rp(work_c(1)))
                    deallocate(work_c)
                    allocate(work_c(lwork))

                    ! calculate eigenvalues
                    call zhseqr('E','N',n_mod_x,1,n_mod_x,res_loc,n_mod_x,&
                        &res_ev,0._dp,n_mod_x,work_c,lwork,ierr)
                    chckerr('zhseqr failed')
                    call print_ex_2d(['real part','imag part'],'res_ev',&
                        &reshape([rp(res_ev),ip(res_ev)],[n_mod_x,2]),&
                        &draw=.false.)
                    call draw_ex(['real part','imag part'],'res_ev',2,1,&
                        &.false.)
                    deallocate(work_c)
                    deallocate(res_ev)
                end if
#endif
            end if

#if ldebug
            ! calculate the Eigenvalues of the original real res2
            if (debug_calc_vac_res) then
                ! get size of work matrix required for pdgehrd
                allocate(work(1))
                allocate(tau(2*n_mod_x-1))
                call pdgehrd(2*n_mod_x,1,2*n_mod_x,res2,1,1,desc_res2,tau,work,&
                    &-1,ierr)
                chckerr('pdgehrd failed')
                lwork = nint(work(1))
                deallocate(work)
                allocate(work(lwork))

                ! convert to upper Heisenberg form
                call pdgehrd(2*n_mod_x,1,2*n_mod_x,res2,1,1,desc_res2,tau,&
                    &work,lwork,ierr)
                chckerr('pdgehrd failed')
                deallocate(work)
                deallocate(tau)

                ! calculate eigenvalues
                lwork = 2*n_mod_x / vac%bs
                if (lwork*vac%bs.lt.2*n_mod_x) lwork = lwork + 1
                lwork = 7*lwork / lcm(vac%n_p(1),vac%n_p(2))
                lwork = 3*2*n_mod_x + max(2*2*n_mod_x+2*n_loc(2) , lwork)
                allocate(work(lwork))
                allocate(res2_ev(2*n_mod_x,2))
                call pdlahqr(.false.,.false.,2*n_mod_x,1,2*n_mod_x,res2,&
                    &desc_res2,res2_ev(:,1),res2_ev(:,2),1,2*n_mod_x,[0._dp],&
                    &desc_res2,work,lwork,[0],1,ierr)
                chckerr('pdlahqr failed')
                call print_ex_2d(['real part','imag part'],'res2_ev',&
                    &res2_ev,draw=.false.)
                call draw_ex(['real part','imag part'],'res2_ev',2,1,.false.)
                deallocate(work)
                deallocate(res2_ev)

                ! user output
                call lvl_ud(-1)
                call writo('They should all be positive',persistent=rank_o)
            end if
#endif

            call lvl_ud(-1)
        end if

        call lvl_ud(-1)

        call writo('Done calculating vacuum quantities')
    end function calc_vac_res

    !> Calculate vacuum potential at some positions that are not resonant.
    !!
    !! This  is done  by  using the  relation  between  H and  G  on the  plasma
    !! boundary,
    !!  \f$\overline{\text{H}}_\text{s} \overline{\Phi}_\text{s} =
    !!  \overline{\text{G}}_\text{s} \overline{\text{D}\phi}_\text{s}\f$,
    !! where \f$\overline{\text{D}\phi}_\text{s}\f$ is  the normal derivative of
    !! the potential on the  plasma edge. The matrices \f$\overline{\text{H}}\f$
    !! and \f$\overline{\text{G}}\f$ have to be calculated in advance.
    !!
    !! This  relation is  then  used  at different  positions  to calculate  the
    !! potential at a different position through
    !!  \f$4 \pi \phi = \left[ - \overline{\text{H}} \overline{\text{I}}
```

```
    !!  \overline{\text{H}}_\text{s}^{-1} \overline{\text{G}}_\text{s}  +
    !!  \overline{\text{G}} \overline{\text{I}} \right]
    !!  \overline{\text{D}\phi}_\text{s}\f$,
    !! where \f$\overline{\text{I}}\f$ is an integration rule.
    !!
    !! This  system  of equations  contains  one  row  per  point at  which  the
    !! potential is to be calculated.
    !!
    !! Currently, this  routine creates a  3-D regular grid that  is equidistant
    !! in  cylindrical  coordinates, with  \c  n_vac_plot  values  in R  and  Z,
    !! and  \c  n_zeta_plot values  in  the  cylindrical  angle. The  limits  in
    !! these  variables,   furthermore,  are  given  by   \c  min_Rvac_plot,  \c
    !! max_Rvac_plot, \c  min_Zvac_plot, \c max_Zvac_plot, \c  min_zeta_plot and
    !! \c max_zeta_plot.
    integer function vac_pot_plot(grid_sol,vac,sol,X_id) result(ierr)
        use num_vars, only: min_zeta_plot, max_zeta_plot, min_rvac_plot, &
            &max_rvac_plot, min_zvac_plot, max_zvac_plot, n_vac_plot, &
            &n_zeta_plot, n_procs, rank, use_pol_flux_f
        use sol_vars, only: sol_type
        use mpi_utilities, only: broadcast_var
        use iso_c_binding
        use vac_vars, only: in_context
        use vac_utilities, only: mat_dis2loc
        use x_vars, only: prim_x, n_mod_x
        use vac_vars, only: set_loc_lims

        ! input / output
        type(grid_type), intent(in) :: grid_sol                                 !< solution grid
        type(vac_type), intent(inout) :: vac                                    !< vacuum variables
        type(sol_type), intent(in) :: sol                                       !< solution variables
        integer, intent(in) :: x_id                                             !< nr. of Eigenvalue

        ! local variables
        integer :: id, jd, kd                                                   ! counters
        integer :: rd, cd                                                       ! global counter for row and col
        integer :: i_rd, i_cd                                                   ! index of subrow and subcol
        integer :: rdl, cdl                                                     ! local counter for row and col
        integer :: desc_rhs(blacsctxtsize)                                      ! descriptor for RHS
        integer :: desc_gh_loc(blacsctxtsize)                                   ! descriptor for local G and H
        integer :: desc_phi(blacsctxtsize)                                      ! descriptor for Phi
        integer :: n_row_plot                                                   ! n_loc(1) for n_vac_plot rows
        integer :: n_col_2                                                      ! n_loc(2) for 2 columns (real and imaginary part)
        integer :: lims_cl(2)                                                   ! lims_c_loc in local coordinates
        integer, allocatable :: lims_r_loc(:,:)                                 ! local column limits for n_row_plot
        integer, allocatable :: lims_c_loc(:,:)                                 ! local column limits for n_col_2
        integer :: sec_x_loc                                                    ! local sec_X
        integer :: fac_x(2)                                                     ! (n,-m) for poloidal or (-m,n) for toroidal flux
        real(dp) :: zeta_loc                                                    ! local zeta
        real(dp), allocatable :: x_vec(:,:)                                     ! x_vec of influence points
        real(dp), allocatable :: g_loc(:,:)                                     ! local G
        real(dp), allocatable :: h_loc(:,:)                                     ! local H
        real(dp), allocatable :: rhs(:,:,:)                                     ! 2 right-hand sides, real and complex
        real(dp), allocatable :: phi(:,:)                                       ! Phi
        real(dp), allocatable :: phi_ser(:,:)                                   ! serial version of Phi
        real(dp), allocatable :: phi_2d(:,:)                                    ! 2-D serial version of Phi
        complex(dp), allocatable :: sol_vec(:)                                  ! solution vector at edge
        complex(dp) :: rhs_complex                                              ! complex local RHS
        character(len=max_str_ln) :: err_msg                                    ! error message
#if ldebug
        real(dp), allocatable :: rhs_loc(:,:)                                   ! local version of RHS
        !character(len=max_str_ln) :: plot_title                                 ! plot title
        !character(len=max_str_ln) :: plot_name                                  ! plot name
#endif

        character(*), parameter :: rout_name = 'vac_pot_plot'

        ! initialize ierr
        ierr = 0

        ! distribute solution vector
        allocate(sol_vec(n_mod_x))
        if (rank.eq.n_procs-1) sol_vec = sol%vec(:,grid_sol%loc_n_r,x_id)
        ierr = broadcast_var(sol_vec,n_procs-1)
        chckerr('')

        ! calculate matrices G and H
        ierr = calc_gh(vac)
        chckerr('')

        ! Only for processes that are in the blacs context
        if (in_context(vac%ctxt_HG)) then
            ! set sizes
            n_row_plot = numroc(product(n_vac_plot),vac%bs,vac%ind_p(1),0,&
                &vac%n_p(1))
            n_col_2 = numroc(2,vac%bs,vac%ind_p(2),0,vac%n_p(2))
```

```
            call set_loc_lims(n_row_plot,vac%bs,vac%ind_p(1),vac%n_p(1),&
                &lims_r_loc)
            call set_loc_lims(n_col_2,vac%bs,vac%ind_p(2),vac%n_p(2),&
                &lims_c_loc)

            ! allocate RHS and set descriptor
            allocate(rhs(vac%n_loc(1),n_col_2,2))
            rhs = 0._dp
            call descinit(desc_rhs,vac%n_bnd,2,vac%bs,vac%bs,0,0,&
                &vac%ctxt_HG,max(1,vac%n_loc(1)),ierr)
            chckerr('descinit failed for RHS')

            ! initialize the right hand sides as the normal derivative of Phi at
            ! the plasma edge
            subcols: do i_cd = 1,size(lims_c_loc,2)
                col: do cd = lims_c_loc(1,i_cd),lims_c_loc(2,i_cd)
                    ! set local column index
                    cdl = sum(lims_c_loc(2,1:i_cd-1)-&
                        &lims_c_loc(1,1:i_cd-1)+1) + &
                        &cd-lims_c_loc(1,i_cd)+1

                    ! iterate over all subrows of matrix with n_bnd rows
                    subrows: do i_rd = 1,size(vac%lims_r,2)
                        row: do rd = vac%lims_r(1,i_rd),vac%lims_r(2,i_rd)
                            ! set local row index
                            rdl = sum(vac%lims_r(2,1:i_rd-1)-&
                                &vac%lims_r(1,1:i_rd-1)+1) + &
                                &rd-vac%lims_r(1,i_rd)+1

                            ! sum over all modes to get parallel derivative of X
                            do jd = 1,n_mod_x
                                sec_x_loc = vac%sec_X(jd)
                                if (use_pol_flux_f) then
                                    fac_x = [prim_x,-sec_x_loc]
                                else
                                    ierr = 1
                                    err_msg = 'Currently must use poloidal flux'
                                    chckerr(err_msg)
                                    !fac_X = [-prim_X,sec_X_loc]
                                end if

                                rhs_complex = iu*(fac_x(1)*vac%jq+fac_x(2))*&
                                    &exp(iu*fac_x(2)*vac%ang(rd,1))*sol_vec(jd)
                                if (cd.eq.1) rhs(rdl,cdl,:) = &
                                    &rhs(rdl,cdl,:) + rp(rhs_complex)
                                if (cd.eq.2) rhs(rdl,cdl,:) = &
                                    &rhs(rdl,cdl,:) + ip(rhs_complex)
                            end do
                        end do row
                    end do subrows
                end do col
            end do subcols

            ! solve for Phi and save in second index of RHS
            ierr = solve_phi_bem(vac,rhs(:,:,1),rhs(:,:,2),[vac%n_bnd,2],&
                &[vac%n_loc(1),n_col_2],lims_c_loc,desc_rhs)
            chckerr('')

#if ldebug
            ! get serial version on last process
            allocate(rhs_loc(vac%n_bnd,2))
            do jd = 1,2
                ierr = mat_dis2loc(vac%ctxt_HG,rhs(:,:,jd),vac%lims_r,&
                    &lims_c_loc,rhs_loc,proc=n_procs-1)
                chckerr('')
                if (rank.eq.n_procs-1) then
                    call print_ex_2d(['dphi','phi '],'RHS_'//trim(i2str(jd)),&
                        &rhs_loc,x=vac%ang(:,1:1),draw=.false.)
                    call draw_ex(['dphi','phi '],'RHS_'//trim(i2str(jd)),2,1,&
                        &.false.)
                end if
            end do
            deallocate(rhs_loc)
#endif

            ! convert RHS into IRHS where I contains the integration rule:
            !         (θ_{i+1}-θ_{i-1})/2   for i = 2..n-1
            !   I_i = (θ_2-θ_1}/2           for i = 1
            !         (θ_{n}-θ_{n-1}}/2     for i = n
            ! with n = vac%n_bnd.
            subrows2: do i_rd = 1,size(vac%lims_r,2)
                row2: do rd = vac%lims_r(1,i_rd),vac%lims_r(2,i_rd)
                    ! set local row index
                    rdl = sum(vac%lims_r(2,1:i_rd-1)-&
                        &vac%lims_r(1,1:i_rd-1)+1) + &
                        &rd-vac%lims_r(1,i_rd)+1

```

```
                    subcols2: do i_cd = 1,size(lims_c_loc,2)
                        ! local limits
                        lims_cl = sum(lims_c_loc(2,1:i_cd-1)-&
                            &lims_c_loc(1,1:i_cd-1)+1) + &
                            &lims_c_loc(:,i_cd)-lims_c_loc(1,i_cd)+1

                        rhs(rdl,lims_cl(1):lims_cl(2),:) = &
                            &rhs(rdl,lims_cl(1):lims_cl(2),:) * 0.5_dp*&
                            &(vac%ang(min(rd+1,vac%n_bnd),1)-&
                            &vac%ang(max(rd-1,1),1))
                    end do subcols2
                end do row2
            end do subrows2

            ! allocate G, H and x_vec and set descriptor
            allocate(g_loc(n_row_plot,vac%n_loc(2)))
            allocate(h_loc(n_row_plot,vac%n_loc(2)))
            allocate(x_vec(product(n_vac_plot),2))
            g_loc = 0._dp
            h_loc = 0._dp
            call descinit(desc_gh_loc,product(n_vac_plot),vac%n_bnd,vac%bs,&
                &vac%bs,0,0,vac%ctxt_HG,max(1,n_row_plot),ierr)
            chckerr('descinit failed for GH_loc')

            ! loop over all toroidal points
            do kd = 1,n_zeta_plot
                ! set local zeta
                zeta_loc = min_zeta_plot + (max_zeta_plot-min_zeta_plot) * &
                    &(kd-1._dp)/max(1._dp,n_zeta_plot-1._dp)

                ! set up variables to calculate local G and H
                row3: do rd = 1,product(n_vac_plot)
                    ! set up 2-D index for x_vec
                    id = mod(rd-1,n_vac_plot(1))+1
                    jd = (rd-1)/n_vac_plot(1)+1

                    ! set up x_vec
                    x_vec(rd,1) = min_rvac_plot + &
                        &(max_rvac_plot-min_rvac_plot)*&
                        &(id-1._dp)/(n_vac_plot(1)-1._dp)
                    x_vec(rd,2) = min_zvac_plot + &
                        &(max_zvac_plot-min_zvac_plot)*&
                        &(jd-1._dp)/(n_vac_plot(2)-1._dp)
                end do row3
                !call print_ex_2D(['R','Z'],'x_vec_'//trim(i2str(rank)),x_vec,draw=.false.)
                !call draw_ex(['R','Z'],'x_vec_'//trim(i2str(rank)),2,1,.false.)

                ! calc local G and H
                ierr = calc_gh(vac,n_r_in=product(n_vac_plot),&
                    &lims_r_in=lims_r_loc,x_vec_in=x_vec,g_in=g_loc,h_in=h_loc)
                chckerr('')

#if ldebug
                !if (debug_vac_pot_plot) then
                    !subrows4: do i_rd = 1,size(lims_r_loc,2)
                        !row4: do rd = lims_r_loc(1,i_rd),lims_r_loc(2,i_rd)
                            !subcols4: do i_cd = 1,size(vac%lims_c,2)
                                !! output
                                !plot_title = 'for row '//trim(i2str(rd))//&
                                    !&' and starting column '//&
                                    !&trim(i2str(vac%lims_c(1,i_cd)))
                                !plot_name = '_'//trim(i2str(kd))//'_'//&
                                    !&trim(i2str(rd))//'_'//&
                                    !&trim(i2str(vac%lims_c(1,i_cd)))

                                !! local limits
                                !lims_cl = sum(vac%lims_c(2,1:i_cd-1)-&
                                    !&vac%lims_c(1,1:i_cd-1)+1) + &
                                    !&vac%lims_c(:,i_cd)-vac%lims_c(1,i_cd)+1

                                !! output
                                !call print_ex_2D('G '//trim(plot_title),&
                                    !&'G_loc'//trim(plot_name),&
                                    !&G_loc(rdl,lims_cl(1):lims_cl(2)),x=vac%ang&
                                    !&(vac%lims_c(1,i_cd):vac%lims_c(2,i_cd),1),&
                                    !&draw=.false.)
                                !call draw_ex(['G '//trim(plot_title)],&
                                    !&'G_loc'//trim(plot_name),1,1,.false.)
                                !call print_ex_2D('H '//trim(plot_title),&
                                    !&'H_loc'//trim(plot_name),&
                                    !&H_loc(rdl,lims_cl(1):lims_cl(2)),x=vac%ang&
                                    !&(vac%lims_c(1,i_cd):vac%lims_c(2,i_cd),1),&
                                    !&draw=.false.)
                                !call draw_ex(['H '//trim(plot_title)],&
                                    !&'H_loc'//trim(plot_name),1,1,.false.)
                            !end do subcols4
                        !end do row4
```

```
                    !end do subrows4
                !end if
#endif

                ! multiply G with RHS(1) and save in Phi
                allocate(phi(n_row_plot,n_col_2))
                call descinit(desc_phi,product(n_vac_plot),2,vac%bs,&
                    &vac%bs,0,0,vac%ctxt_HG,max(1,n_row_plot),ierr)
                chckerr('descinit failed for Phi')
                call pdgemm('N','N',product(n_vac_plot),2,vac%n_bnd,1._dp,&
                    &g_loc,1,1,desc_gh_loc,rhs(:,:,1),1,1,desc_rhs,0._dp,&
                    &phi,1,1,desc_phi)

                ! add -H multiplied by RHS(2)
                call pdgemm('N','N',product(n_vac_plot),2,vac%n_bnd,-1._dp,&
                    &h_loc,1,1,desc_gh_loc,rhs(:,:,2),1,1,desc_rhs,1._dp,&
                    &phi,1,1,desc_phi)

                ! get serial version on last process
                allocate(phi_ser(product(n_vac_plot),2))
                ierr = mat_dis2loc(vac%ctxt_HG,phi,lims_r_loc,&
                    &lims_c_loc,phi_ser,proc=n_procs-1)
                chckerr('')
#if ldebug
                if (rank.eq.n_procs-1) then
                    call print_ex_2d(['dphi','phi '],'Phi'//trim(i2str(kd)),&
                        &phi_ser,draw=.false.)
                    call draw_ex(['dphi','phi '],'Phi'//trim(i2str(kd)),2,1,&
                        &.false.)
                end if
#endif

                ! multiply with exponential of toroidal angle and take real part
                ! (phi_C = - zeta_F)
                if (rank.eq.n_procs-1) then
                    allocate(phi_2d(n_vac_plot(1),n_vac_plot(2)))
                    phi_2d = -1./(4*pi)*reshape(&
                        &phi_ser(:,1)*cos(vac%prim_X*zeta_loc) + &
                        &phi_ser(:,2)*sin(vac%prim_X*zeta_loc),n_vac_plot)
                    call plot_hdf5('Phi','Phi',reshape(phi_2d,[n_vac_plot,1]),&
                        &x=cos(vac%prim_X*zeta_loc)*&
                        &reshape(x_vec(:,1),[n_vac_plot,1]),&
                        &y=sin(vac%prim_X*zeta_loc)*&
                        &reshape(x_vec(:,1),[n_vac_plot,1]),&
                        &z=reshape(x_vec(:,2),[n_vac_plot,1]))
                end if
                deallocate(x_vec)
            end do
        end if
    end function vac_pot_plot

    !> \public Calculates potential  \c Phi on boundary of BEM  in terms of some
    !! right-hand side .
    !!
    !! This is done by solving with STRUMPack the system of equations
    !!  \f$\overline{\text{H}}\overline{\Phi} =
    !!  \overline{\text{G}}\overline{\text{R}}\f$,
    !! where \f$\overline{\text{R}}\f$ is the right-hand side, for example equal
    !! to \f$\overline{\text{E}}\overline{\text{P}}\f$  to solve in  function of
    !! the individual Fourier modes.
    integer function solve_phi_bem(vac,R,Phi,n_RPhi,n_loc_RPhi,lims_c_RPhi,&
        &desc_RPhi) result(ierr)

        use vac_vars, only: in_context
#if ldebug
        use grid_vars, only: min_par_x, max_par_x
        use rich_vars, only: n_par_x
#endif

        character(*), parameter :: rout_name = 'solve_Phi_BEM'

        ! input / output
        type(vac_type), intent(in), target :: vac                          !< vacuum variables
        real(dp), intent(inout), target :: phi(:,:)                        !< \f$\overline{\Phi}\f$
        real(dp), intent(in) :: r(:,:)                                     !< \f$\overline{\text{R}}\f$
        integer, intent(in) :: n_rphi(2)                                   !< \c n for \f$\overline{\text{R}}\f$ and \f$\overline{\Phi}\f$
        integer, intent(in) :: n_loc_rphi(2)                               !< local \c n for \f$\overline{\text{R}}\f$ and \f$\overline{\Phi}\f$
        integer, intent(in) :: lims_c_rphi(:,:)                            !< local limits for row and columns of \f$\overline{\text{R}}\f$ and \f$\overline{\Phi}\f$
        integer, intent(in), target :: desc_rphi(blacsctxtsize)            !< descriptor for \f$\overline{\text{R}}\f$ and \f$\overline{\Phi}\f$

        ! local variables
        integer, target :: desc_gr(blacsctxtsize)                          ! descriptor for GR
        real(dp) :: sdp_err                                                ! error
```

```
        real(dp), allocatable, target :: gr(:,:)                                ! GR
        type(c_ptr) :: cdesch, ch                                               ! C pointers to descriptor of H and H itself
        type(c_ptr) :: cdescgr, cgr                                             ! C pointers to descriptor of GR and GR itself
        type(c_ptr) :: cdescphi, cphi                                           ! C pointers to descriptor of Phi and Phi itself
        type(strumpackdensepackage_f90_double) :: sdp_loc                       ! Strumpack object for local solve H Phi = GR
#if ldebug
        integer :: lims_rl(2)                                                   ! vac%lims_r in local coordinates
        integer :: id                                                           ! counter
        integer :: cd                                                           ! global counter for row and col
        integer :: cdl                                                          ! local counter for col
        integer :: i_rd, i_cd                                                   ! index of subrow and subcol
        character(len=max_str_ln) :: plot_title                                 ! plot title
        character(len=max_str_ln) :: plot_name                                  ! plot name
        real(dp), allocatable :: ang(:)                                         ! angle
#endif

        ! initialize ierr
        ierr = 0

        ! Only for processes that are in the blacs context
        if (in_context(vac%ctxt_HG)) then
            ! allocate auxililary matrices
            allocate(gr(vac%n_loc(1),n_loc_rphi(2)))

            ! initialize descriptors
            call descinit(desc_gr,vac%n_bnd,n_rphi(2),vac%bs,vac%bs,0,0,&
                &vac%ctxt_HG,max(1,vac%n_loc(1)),ierr)
            chckerr('descinit failed for GR')

            ! calculate GR
            call pdgemm('N','N',vac%n_bnd,n_rphi(2),vac%n_bnd,1._dp,vac%G,1,1,&
                &vac%desc_G,r,1,1,desc_rphi,0._dp,gr,1,1,desc_gr)

            ! set C pointers
            ch = c_loc(vac%H)
            cdesch = c_loc(vac%desc_H)
            cgr = c_loc(gr)
            cdescgr = c_loc(desc_gr)
            cphi = c_loc(phi)
            cdescphi = c_loc(desc_rphi)

            ! initialize the solver and set parameters
            call sdp_f90_double_init(sdp_loc,vac%MPI_Comm)
            sdp_loc%use_HSS=1
            sdp_loc%levels_HSS=4
            sdp_loc%min_rand_HSS=10
            sdp_loc%lim_rand_HSS=5
            sdp_loc%inc_rand_HSS=10
            sdp_loc%max_rand_HSS=100
            sdp_loc%tol_HSS=1d-12
            sdp_loc%steps_IR=10
            sdp_loc%tol_IR=1d-10

            ! compression
            call sdp_f90_double_compress_a(sdp_loc,ch,cdesch)

#if ldebug
            ! accuracy checking
            sdp_err = sdp_f90_double_check_compression(sdp_loc,ch,cdesch)
#endif

            ! factorization
            call sdp_f90_double_factor(sdp_loc,ch,cdesch)

            ! solve H Phi = G R
            call sdp_f90_double_solve(sdp_loc,cphi,cdescphi,cgr,cdescgr)

#if ldebug
            ! accuracy checking
            sdp_err = sdp_f90_double_check_solution(sdp_loc,ch,cdesch,cphi,&
                &cdescphi,cgr,cdescgr)
#endif

            ! iterative refinement
            sdp_err = sdp_f90_double_refine(sdp_loc,ch,cdesch,cphi,cdescphi,&
                &cgr,cdescgr)

#if ldebug
            ! statistics
            call sdp_f90_double_print_statistics(sdp_loc)

            ! user output
            subcols: do i_cd = 1,size(lims_c_rphi,2)
                col: do cd = lims_c_rphi(1,i_cd),lims_c_rphi(2,i_cd)
```

```
                    ! set local column index
                    cdl = sum(lims_c_rphi(2,1:i_cd-1)-&
                        &lims_c_rphi(1,1:i_cd-1)+1) + &
                        &cd-lims_c_rphi(1,i_cd)+1

                    subrows2: do i_rd = 1,size(vac%lims_r,2)
                        ! set local row limits
                        lims_rl = sum(vac%lims_r(2,1:i_rd-1)-&
                            &vac%lims_r(1,1:i_rd-1)+1) + &
                            &vac%lims_r(:,i_rd)-vac%lims_r(1,i_rd)+1

                        ! set angle
                        allocate(ang(vac%lims_r(2,i_rd)-vac%lims_r(1,i_rd)+1))
                        select case (vac%style)
                            case (1)                                        ! field-line 3-D
                                do id = 1,vac%n_ang(1)
                                    ang(id:&
                                        &id+vac%n_ang(1)*(vac%n_ang(2)-1):&
                                        &vac%n_ang(1)) = &
                                        &pi * (min_par_x + (id-1)*&
                                        &(max_par_x-min_par_x)/(n_par_x-1))
                                end do
                            case (2)                                        ! axisymmetric
                                ang = reshape(vac%ang,[vac%n_bnd])
                        end select

                        ! output
                        plot_title = 'for column '//trim(i2str(cd))//&
                            &' and starting row '//&
                            &trim(i2str(vac%lims_r(1,i_rd)))
                        plot_name = '_'//trim(i2str(cd))//'_'//&
                            &trim(i2str(vac%lims_r(1,i_rd)))
                        call print_ex_2d('Phi '//trim(plot_title),&
                            &'Phi'//trim(plot_name),&
                            &phi(lims_rl(1):lims_rl(2),cdl),&
                            &x=ang(vac%lims_r(1,i_rd):vac%lims_r(2,i_rd)),&
                            &draw=.false.)
                        call draw_ex(['Phi '//trim(plot_title)],&
                            &'Phi'//trim(plot_name),1,1,.false.)

                        ! clean up
                        deallocate(ang)
                    end do subrows2
                end do col
            end do subcols
#endif

            ! destroy Strumpack object
            call sdp_f90_double_destroy(sdp_loc)
        end if
    end function solve_phi_bem

    !> Print vacuum quantities of a certain order to an output file
    !!
    !!  - vac:
    !!      - \c misc_vac
    !!      - \c norm
    !!      - \c x_vec
    !!      - \c vac_res
    !!
    !! If \c  rich_lvl is provided, \c  "_R_[rich_lvl]" is appended to  the data
    !! name if it is \c >0.
    !!
    !! \note
    !!  -#  This routine  should  be called  by a  single  process, in  contrast
    !!  to  the   other  output   routines  such   as  eq_ops.print_output_eq(),
    !!  print_output_sol(), ...
    !!  -# This  process should  be the last  one, as it  will set  the boundary
    !!  contribution.
    !!  -# This routine  does not save the  \c H and \c G  matrices because they
    !!  are large and foreseen to be  saved directly as Hierarchical matrices in
    !!  the future. They therefore need  to be regenerated if Richardson restart
    !!  is done, or  when after a jump to solution,  another Richardson level is
    !!  attempted. In calc_vac, it is  checked when copying the vacuum variables
    !!  from previous  to current Richardson  level, whether the  \c G and  \c H
    !!  matrices are allocated and if not, they are calculated.
    !!
    !! \return ierr
    integer function print_output_vac(vac,data_name,rich_lvl) result(ierr)
        use x_vars, only: n_mod_x
        use num_vars, only: pb3d_name
        use hdf5_ops, only: print_hdf5_arrs
        use hdf5_vars, only: dealloc_var_1d, var_1d_type, &
            &max_dim_var_1d

        character(*), parameter :: rout_name = 'print_output_vac'

```

```
        ! input / output
        type(vac_type), intent(in) :: vac                                   !< vacuum variables
        character(len=*), intent(in) :: data_name                           !< name under which to store
        integer, intent(in), optional :: rich_lvl                           !< Richardson level to print

        ! local variables
        type(var_1d_type), allocatable, target :: vac_1d(:)                 ! 1D equivalent of vac variables
        type(var_1d_type), pointer :: vac_1d_loc => null()                  ! local element in vac_1D
        integer :: id                                                       ! counter

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Write vacuum variables to output file')
        call lvl_ud(1)

        ! Set up the 1D equivalents of the perturbation variables
        allocate(vac_1d(max_dim_var_1d))

        ! set up variables vac_1D
        id = 1

        ! misc_vac
        vac_1d_loc => vac_1d(id); id = id+1
        vac_1d_loc%var_name = 'misc_vac'
        allocate(vac_1d_loc%tot_i_min(1),vac_1d_loc%tot_i_max(1))
        allocate(vac_1d_loc%loc_i_min(1),vac_1d_loc%loc_i_max(1))
        vac_1d_loc%loc_i_min = [1]
        vac_1d_loc%loc_i_max = [6]
        vac_1d_loc%tot_i_min = vac_1d_loc%loc_i_min
        vac_1d_loc%tot_i_max = vac_1d_loc%loc_i_max
        allocate(vac_1d_loc%p(6))
        vac_1d_loc%p = [vac%style*1._dp,vac%n_bnd*1._dp,vac%prim_X*1._dp,&
            &vac%n_ang*1._dp,vac%jq]

        ! sec_X
        vac_1d_loc => vac_1d(id); id = id+1
        vac_1d_loc%var_name = 'sec_X'
        allocate(vac_1d_loc%tot_i_min(1),vac_1d_loc%tot_i_max(1))
        allocate(vac_1d_loc%loc_i_min(1),vac_1d_loc%loc_i_max(1))
        vac_1d_loc%tot_i_min = [1]
        vac_1d_loc%tot_i_max = n_mod_x
        vac_1d_loc%loc_i_min = vac_1d_loc%tot_i_min
        vac_1d_loc%loc_i_max = vac_1d_loc%tot_i_max
        allocate(vac_1d_loc%p(n_mod_x))
        vac_1d_loc%p = vac%sec_X*1._dp

        ! norm
        vac_1d_loc => vac_1d(id); id = id+1
        vac_1d_loc%var_name = 'norm'
        allocate(vac_1d_loc%tot_i_min(2),vac_1d_loc%tot_i_max(2))
        allocate(vac_1d_loc%loc_i_min(2),vac_1d_loc%loc_i_max(2))
        vac_1d_loc%tot_i_min = [1,1]
        vac_1d_loc%tot_i_max = shape(vac%norm)
        vac_1d_loc%loc_i_min = vac_1d_loc%tot_i_min
        vac_1d_loc%loc_i_max = vac_1d_loc%tot_i_max
        allocate(vac_1d_loc%p(size(vac%norm)))
        vac_1d_loc%p = reshape(vac%norm,[size(vac%norm)])

        ! x_vec
        vac_1d_loc => vac_1d(id); id = id+1
        vac_1d_loc%var_name = 'x_vec'
        allocate(vac_1d_loc%tot_i_min(2),vac_1d_loc%tot_i_max(2))
        allocate(vac_1d_loc%loc_i_min(2),vac_1d_loc%loc_i_max(2))
        vac_1d_loc%tot_i_min = [1,1]
        vac_1d_loc%tot_i_max = shape(vac%x_vec)
        vac_1d_loc%loc_i_min = vac_1d_loc%tot_i_min
        vac_1d_loc%loc_i_max = vac_1d_loc%tot_i_max
        allocate(vac_1d_loc%p(size(vac%x_vec)))
        vac_1d_loc%p = reshape(vac%x_vec,[size(vac%x_vec)])

        ! RE_res
        vac_1d_loc => vac_1d(id); id = id+1
        vac_1d_loc%var_name = 'RE_res'
        allocate(vac_1d_loc%tot_i_min(2),vac_1d_loc%tot_i_max(2))
        allocate(vac_1d_loc%loc_i_min(2),vac_1d_loc%loc_i_max(2))
        vac_1d_loc%tot_i_min = [1,1]
        vac_1d_loc%tot_i_max = [n_mod_x,n_mod_x]
        vac_1d_loc%loc_i_min = vac_1d_loc%tot_i_min
        vac_1d_loc%loc_i_max = vac_1d_loc%tot_i_max
        allocate(vac_1d_loc%p(size(vac%res)))
        vac_1d_loc%p = reshape(rp(vac%res),[size(vac%res)])

        ! IM_res
        vac_1d_loc => vac_1d(id); id = id+1
        vac_1d_loc%var_name = 'IM_res'
```

```
        allocate(vac_1d_loc%tot_i_min(2),vac_1d_loc%tot_i_max(2))
        allocate(vac_1d_loc%loc_i_min(2),vac_1d_loc%loc_i_max(2))
        vac_1d_loc%tot_i_min = [1,1]
        vac_1d_loc%tot_i_max = [n_mod_x,n_mod_x]
        vac_1d_loc%loc_i_min = vac_1d_loc%tot_i_min
        vac_1d_loc%loc_i_max = vac_1d_loc%tot_i_max
        allocate(vac_1d_loc%p(size(vac%res)))
        vac_1d_loc%p = reshape(ip(vac%res),[size(vac%res)])

        ! copy variables specific to style
        select case (vac%style)
            case (1)                                                    ! field-line 3-D
                ! h_fac
                vac_1d_loc => vac_1d(id); id = id+1
                vac_1d_loc%var_name = 'h_fac'
                allocate(vac_1d_loc%tot_i_min(2),vac_1d_loc%tot_i_max(2))
                allocate(vac_1d_loc%loc_i_min(2),vac_1d_loc%loc_i_max(2))
                vac_1d_loc%tot_i_min = [1,1]
                vac_1d_loc%tot_i_max = shape(vac%h_fac)
                vac_1d_loc%loc_i_min = vac_1d_loc%tot_i_min
                vac_1d_loc%loc_i_max = vac_1d_loc%tot_i_max
                allocate(vac_1d_loc%p(size(vac%h_fac)))
                vac_1d_loc%p = reshape(vac%h_fac,[size(vac%h_fac)])
            case (2)                                                    ! axisymmetric
                ! ang
                vac_1d_loc => vac_1d(id); id = id+1
                vac_1d_loc%var_name = 'ang'
                allocate(vac_1d_loc%tot_i_min(2),vac_1d_loc%tot_i_max(2))
                allocate(vac_1d_loc%loc_i_min(2),vac_1d_loc%loc_i_max(2))
                vac_1d_loc%tot_i_min = [1,1]
                vac_1d_loc%tot_i_max = shape(vac%ang)
                vac_1d_loc%loc_i_min = vac_1d_loc%tot_i_min
                vac_1d_loc%loc_i_max = vac_1d_loc%tot_i_max
                allocate(vac_1d_loc%p(size(vac%ang)))
                vac_1d_loc%p = reshape(vac%ang,[size(vac%ang)])

                ! dnorm
                vac_1d_loc => vac_1d(id); id = id+1
                vac_1d_loc%var_name = 'dnorm'
                allocate(vac_1d_loc%tot_i_min(2),vac_1d_loc%tot_i_max(2))
                allocate(vac_1d_loc%loc_i_min(2),vac_1d_loc%loc_i_max(2))
                vac_1d_loc%tot_i_min = [1,1]
                vac_1d_loc%tot_i_max = shape(vac%dnorm)
                vac_1d_loc%loc_i_min = vac_1d_loc%tot_i_min
                vac_1d_loc%loc_i_max = vac_1d_loc%tot_i_max
                allocate(vac_1d_loc%p(size(vac%dnorm)))
                vac_1d_loc%p = reshape(vac%dnorm,[size(vac%dnorm)])
        end select

        ! write
        ierr = print_hdf5_arrs(vac_1d(1:id-1),pb3d_name,trim(data_name),&
            &rich_lvl=rich_lvl,ind_print=.true.)
        chckerr('')

        ! clean up
        call dealloc_var_1d(vac_1d)
        nullify(vac_1d_loc)

        ! user output
        call lvl_ud(-1)
    end function print_output_vac
end module vac_ops
```

D.88 /Users/toonweyens/Code/PB3D/Modules/vac_utilities.f90 File Reference

Modules

- module vac_utilities

  Numerical utilities related to the vacuum response.

Functions/Subroutines

- subroutine, public vac_utilities::calc_gh_int_1 (g, h, x_s, x_in, norm_s, h_fac_in, step_size, tol)

Calculate G_ij and H_ij on an interval for field line aligned configurations.

- subroutine, public vac_utilities::calc_gh_int_2 (g, h, t_s, t_in, x_s, x_in, norm_s, norm_in, aij, ql, tol, b_coeff, n_tor)

  Calculate G_ij and H_ij on an interval for axisymmetric configurations.
- integer function, public vac_utilities::vec_dis2loc (ctxt, vec_dis, lims, vec_loc, proc)

  Gathers a distributed vector on a single process.
- integer function, public vac_utilities::mat_dis2loc (ctxt, mat_dis, lims_r, lims_c, mat_loc, proc)

  Gathers a distributed vector on a single process.
- integer function, public vac_utilities::interlaced_vac_copy (vac_old, vac)

  Copies vacuum variables of a previous level into the appropriate, interlaced place of a next level.

D.89 vac_utilities.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Numerical utilities related to the vacuum response.
!!
!! \see vac_ops for information.
!-----------------------------------------------------------------------------!
module vac_utilities
#include <PB3D_macros.h>
#include <wrappers.h>
    use strumpackdensepackage
    use str_utilities
    use messages
    use output_ops
    use num_vars, only: dp, pi, max_str_ln, iu
    use vac_vars, only: vac_type

    implicit none
    private
    public calc_gh_int_1, calc_gh_int_2, vec_dis2loc, mat_dis2loc, &
        &interlaced_vac_copy

contains
    !> Calculate  G_ij  and  H_ij  on  an  interval  for  field  line  aligned
    !! configurations.
    !!
    !! The indices for the  source variables are <tt>[left:right,dim]</tt> where
    !! \c dim is the Cartesian dimension. The same holds for \c ql.
    !!
    !! For  subintegrals  close  to  the singularity,  the  procedure  uses  the
    !! analytical approximation.
    !!
    !! \note This routine does not calculate the contribution \f$2\beta\f$.
    subroutine calc_gh_int_1(G,H,x_s,x_in,norm_s,h_fac_in,step_size,tol)
        ! input / output
        real(dp), intent(inout) :: g(2)                                         !< G
        real(dp), intent(inout) :: h(2)                                         !< H
        real(dp), intent(in) :: x_s(2,3)                                        !< position vector at left and right limit of source interval
        real(dp), intent(in) :: x_in(3)                                         !< position vector at which to calculate influence
        real(dp), intent(in) :: norm_s(2,3)                                     !< normal vector at left and right limit of source interval
        real(dp), intent(in) :: h_fac_in(4)                                     !< metric factors at which to calculate influence
        real(dp), intent(in) :: step_size(2)                                    !< step sizes in parallel direction and alpha
        real(dp), intent(in) :: tol                                             !< tolerance on distance between points

        ! local variables
        integer :: kd                                                           ! counter
        real(dp) :: r2(2)                                                       ! distance between source and influence points
        real(dp) :: e_fac                                                       ! factor E
        real(dp) :: sineps                                                      ! sin(eps)
        real(dp) :: dum_fac(2)                                                  ! dummy factors

        ! initialize
        g = 0._dp
        h = 0._dp

        ! calculate distance between source points
```

```
        do kd = 1,2
            r2(kd) = sum((x_s(kd,:)-x_in)**2)
        end do

        ! check for (near-)singular point
        if (minval(r2).le.tol) then
            e_fac = sqrt(h_fac_in(1)/h_fac_in(3))*step_size(2)/step_size(1)     ! |e_alpha|/|e_theta| d_par_X / d_par_X
            sineps = h_fac_in(2)/sqrt(h_fac_in(1)*h_fac_in(3))
            dum_fac(1) = sqrt(1+2*e_fac*sineps+e_fac**2)
            dum_fac(2) = sqrt(1-2*e_fac*sineps+e_fac**2)
            g = -0.5_dp/sqrt(h_fac_in(1)) * step_size(1) * (&
                &log(abs((dum_fac(2) + e_fac + sineps)/&
                &(dum_fac(1) - e_fac + sineps))) &
                &- e_fac/sineps**2* &
                &log(abs((dum_fac(2) + e_fac + sineps)/&
                &(dum_fac(1) - e_fac + sineps))) &
                &)
            h = h_fac_in(4)*g
        else
            g = -1._dp/sqrt(r2)
            do kd = 1,2
                h(kd) = sum(norm_s(kd,:)*(x_s(kd,:)-x_in))*(-g(kd))**(-3)
            end do
        end if
    end subroutine calc_gh_int_1

    !> Calculate G_ij and H_ij on an interval for axisymmetric configurations.
    !!
    !! The indices for the  source variables are <tt>[left:right,dim]</tt> where
    !! \c dim is the Cartesian dimension. The same holds for \c ql.
    !!
    !! For subintegrals close to the singularity (i.e. where the tolerance is no
    !! being  met), the  routine approximates  the toroidal  functions with  the
    !! analytical approximation.
    !! If  one of  the boundaries  of  the subintegral  is far  enough from  the
    !! singularity, there  is an additional  contribution due to  the difference
    !! between  the  actual  toroidal  function  and  the  approximation.  This
    !! contribution is zero  if both boundaries are close, because  in this case
    !! it  is assumed  that  the  entire integral  is  given  by the  analytical
    !! approximation.
    !!
    !! \note This routine does not calculate the contribution \f$2\beta\f$.
    subroutine calc_gh_int_2(G,H,t_s,t_in,x_s,x_in,norm_s,norm_in,Aij,ql,tol,&
        &b_coeff,n_tor)
        ! input / output
        real(dp), intent(inout) :: g(2)                                         !< G
        real(dp), intent(inout) :: h(2)                                         !< H
        real(dp), intent(in) :: t_s(2)                                          !< pol. angle at left and right limit of source interval
        real(dp), intent(in) :: t_in                                            !< pol. angle at which to calculate influence
        real(dp), intent(in) :: x_s(2,2)                                        !< position vector at left and right limit of source interval
        real(dp), intent(in) :: x_in(2)                                         !< position vector at which to calculate influence
        real(dp), intent(in) :: norm_s(2,2)                                     !< normal vector at left and right limit of source interval
        real(dp), intent(in) :: norm_in(2)                                      !< normal vector at which to calculate influence
        real(dp), intent(in) :: aij(2)                                          !< Aij helper variable for H
        real(dp), intent(in) :: ql(2,2)                                         !< ql for n-1 and n at left and right limit of source interval
        real(dp), intent(in) :: tol                                             !< tolerance on rho2
        real(dp), intent(in) :: b_coeff(2)                                      !< \f$b = \sum_{k=1}^n \frac{2}{2k-1}\f$ for n and n-1
        integer, intent(in) :: n_tor                                            !< toroidal mode number

        ! local variables
        integer :: kd, jd                                                       ! counters
        integer :: kd_bound                                                     ! counter for boundary
        real(dp) :: a_coeff                                                     ! coefficient a = |norm|/8R^2 at influence point
        real(dp) :: t_in_loc                                                    ! local t_in, possibly shifted by 2pi
        real(dp) :: delta_t(2)                                                  ! t_in - t_s
        real(dp) :: rho2(2)                                                     ! square of distance in projected poloidal plane
        real(dp) :: dlogd(2)                                                    ! delta log delta
        real(dp) :: n_loc(2)                                                    ! norm_s / |norm_s|
        real(dp) :: dn_loc(2)                                                   ! d n_loc / dtheta

        ! initialize
        g = 0._dp
        h = 0._dp
        do kd = 1,2
            rho2(kd) = sum((x_s(kd,:)-x_in)**2)
        end do
        if (pi .lt. sum(t_s)/2 - t_in) then                                     ! t_s closer to t_in + 2pi
```

```
            t_in_loc = t_in + 2*pi
        else if (-pi .gt. sum(t_s)/2 - t_in) then                          ! t_s closer to t_in - 2pi
            t_in_loc = t_in - 2*pi
        else                                                               ! t_s closer to t_in
            t_in_loc = t_in
        end if
        n_loc = (norm_s(2,:)/sqrt(sum(norm_s(2,:)**2))+&
            &norm_s(1,:)/sqrt(sum(norm_s(1,:)**2))) / 2._dp
        dn_loc = (norm_s(2,:)/sqrt(sum(norm_s(2,:)**2))-&
            &norm_s(1,:)/sqrt(sum(norm_s(1,:)**2))) / (t_s(2)-t_s(1))

        ! check for (near-)singular point
        if (minval(rho2).le.tol) then
           ! There is at least one singular point: use analytical approach
            a_coeff = sqrt(sum(norm_in**2))/(8._dp*x_in(1)**2)
            do kd = 1,2
                if (kd.eq.1) then
                    delta_t(1) = t_s(1) - t_in_loc
                    delta_t(2) = t_s(2) - t_in_loc
                else
                    delta_t(1) = -(t_s(2) - t_in_loc)
                    delta_t(2) = -(t_s(1) - t_in_loc)
                end if
                do jd = 1,2
                    if (abs(delta_t(jd)).le.0._dp) then
                        dlogd(jd) = 0._dp
                    else
                        dlogd(jd) = log(a_coeff*abs(delta_t(jd)))
                    end if
                end do
                g(kd) = 1._dp/sqrt(x_in(1)*x_s(kd,1)) * (&
                    &b_coeff(2)*(delta_t(2)-delta_t(1)) + &
                    &0.5_dp*delta_t(1) - 1.5_dp*delta_t(2) + &
                    &1._dp/(delta_t(2)-delta_t(1)) * &
                    &(delta_t(1)*(delta_t(1)-2*delta_t(2))*dlogd(1) + &
                    &delta_t(2)**2 * dlogd(2)) &
                    &)
                h(kd) = 0.5*norm_s(kd,1)/x_s(kd,1) * g(kd) + &
                    &(delta_t(2)-delta_t(1)) * (n_tor-0.5_dp)**(-1) * &
                    &aij(kd)/sqrt(x_in(1)*x_s(kd,1))
            end do

            ! Check if  we are at  the boundary  of the analytical  approach. In
            ! this  case there  is  an  extra component  due  to the  difference
            ! between the toroidal function and the analytical approximation.
            ! This  will be  implemented through  a trapezoidal  contribution of
            ! which one side is zero.
            if (maxval(rho2).gt.tol) then
                ! fill G's and H's
                if (rho2(2).gt.rho2(1)) then                               ! right boundary
                    kd_bound = 2
                else                                                       ! left boundary
                    kd_bound = 1
                end if
                g(kd_bound) = g(kd_bound) - 0.5_dp*(t_s(2)-t_s(1))*2._dp/&
                    &sqrt(x_in(1)*x_s(kd_bound,1)) * (ql(kd_bound,2)+&
                    &log(a_coeff*abs(t_s(kd_bound)-t_in_loc))+b_coeff(2))
                h(kd_bound) = h(kd_bound) + 0.5_dp*(t_s(2)-t_s(1))*2._dp/&
                    &sqrt(x_in(1)*x_s(kd_bound,1)) * &
                    &( (-0.5_dp*norm_s(kd_bound,1)/x_s(kd_bound,1) - &
                    &(1._dp+rho2(kd_bound)/(2*x_in(1)*x_s(kd_bound,1)))*&
                    &aij(kd_bound)) * (ql(kd_bound,2)+&
                    &log(a_coeff*abs(t_s(kd_bound)-t_in_loc))+b_coeff(2)) + &
                    &aij(kd_bound) * (ql(kd_bound,1)+&
                    &log(a_coeff*abs(t_s(kd_bound)-t_in_loc))+b_coeff(1)))
            end if
        else
            do kd = 1,2
                ! fill G's and H's
                g(kd) = - (t_s(2)-t_s(1))/sqrt(x_in(1)*x_s(kd,1)) * &
                    &ql(kd,2)
                h(kd) = (t_s(2)-t_s(1))/sqrt(x_in(1)*x_s(kd,1)) * &
                    &( (-0.5_dp*norm_s(kd,1)/x_s(kd,1) - &
                    &(1._dp+rho2(kd)/(2*x_in(1)*x_s(kd,1)))*&
                    &aij(kd))*ql(kd,2) + aij(kd)*ql(kd,1) )
            end do
        end if
    end subroutine calc_gh_int_2

    !> Gathers a distributed vector on a single process.
    !!
    !! For the distributed  vector, the limits of the different  subrows need to
    !! be provided, as discussed in init_vac().
    !!
    !! By  default, the  result lands  on the  master process,  but this  can be
    !! changed  using \c  proc. If  it is  negative, all  processes receive  the
    !! result. In any case, \c vec_loc will be utilized.
```

```
!!
!! \note \c vec_loc needs to be allocated to the total dimensions, even when
!! the results is not received by the process.
integer function vec_dis2loc(ctxt,vec_dis,lims,vec_loc,proc) result(ierr)
    use num_vars, only: n_procs
    use vac_vars, only: in_context

    character(*), parameter :: rout_name = 'vec_dis2loc'

    ! input / output
    integer, intent(in) :: ctxt                                         !< context for vector
    real(dp), intent(in) :: vec_dis(:)                                  !< distributed vector
    integer, intent(in) :: lims(:,:)                                    !< limits for different subrows
    real(dp), intent(inout) :: vec_loc(:)                               !< local vector
    integer, intent(in), optional :: proc                               !< which process receives result

    ! local variables
    integer :: id                                                       ! index of subrow
    integer :: limsl(2)                                                 ! local limits
    integer :: proc_loc                                                 ! local proc
    integer :: proc_dest(2)                                             ! destination process index

    ! initialize ierr
    ierr = 0

    ! select only processes that are in the context
    if (.not.in_context(ctxt)) return

    ! set up destination process index
    proc_loc = 0
    if (present(proc)) proc_loc = proc
    if (proc_loc.gt.0 .and. proc_loc.lt.n_procs) then
        call blacs_pcoord(ctxt,proc_loc,proc_dest(1),proc_dest(2))
    else
        proc_dest = [-1,-1]
    end if

    ! set up total copy of vector
    vec_loc = 0._dp
    do id = 1,size(lims,2)
        ! set local limits
        if (size(vec_dis) .gt. 0) then
            limsl = sum(lims(2,1:id-1)-lims(1,1:id-1)+1) + &
                &lims(:,id)-lims(1,id)+1
            vec_loc(lims(1,id):lims(2,id)) = vec_dis(limsl(1):limsl(2))
        end if
    end do

    ! add them together
    call dgsum2d(ctxt,'all',' ',size(vec_loc),1,vec_loc,size(vec_loc),&
        &proc_dest(1),proc_dest(2))
end function vec_dis2loc

!> Gathers a distributed vector on a single process.
!!
!! \see See vec_dis2loc() for exaplanation.
integer function mat_dis2loc(ctxt,mat_dis,lims_r,lims_c,mat_loc,proc) &
    &result(ierr)
    use num_vars, only: n_procs
    use vac_vars, only: in_context

    character(*), parameter :: rout_name = 'mat_dis2loc'

    ! input / output
    integer, intent(in) :: ctxt                                         !< context for matrix
    real(dp), intent(in) :: mat_dis(:,:)                                !< distributed matrix
    integer, intent(in) :: lims_r(:,:)                                  !< limits for different subrows
    integer, intent(in) :: lims_c(:,:)                                  !< limits for different subcolumns
    real(dp), intent(inout) :: mat_loc(:,:)                             !< local matrix
    integer, intent(in), optional :: proc                               !< which process receives result

    ! local variables
    integer :: i_rd, i_cd                                               ! index of subrow and subcolumn
    integer :: limsl_r(2)                                               ! local row limits
    integer :: limsl_c(2)                                               ! local column limits
    integer :: proc_loc                                                 ! local proc
    integer :: proc_dest(2)                                             ! destination process index

    ! initialize ierr
    ierr = 0

    ! select only processes that are in the context
    if (.not.in_context(ctxt)) return

    ! set up destination process index
    proc_loc = 0
    if (present(proc)) proc_loc = proc
```

```
        if (proc_loc.gt.0 .and. proc_loc.lt.n_procs) then
            call blacs_pcoord(ctxt,proc_loc,proc_dest(1),proc_dest(2))
        else
            proc_dest = [-1,-1]
        end if

        ! set up total copy of matrix
        mat_loc = 0._dp
        do i_rd = 1,size(lims_r,2)
            if (size(mat_dis,1) .gt. 0) then
                limsl_r = sum(&
                    &lims_r(2,1:i_rd-1)-lims_r(1,1:i_rd-1)+1) + &
                    &lims_r(:,i_rd)-lims_r(1,i_rd)+1
                do i_cd = 1,size(lims_c,2)
                    if (size(mat_dis,2) .gt. 0) then
                        limsl_c = sum(&
                            &lims_c(2,1:i_cd-1)-lims_c(1,1:i_cd-1)+1) + &
                            &lims_c(:,i_cd)-lims_c(1,i_cd)+1
                        mat_loc(lims_r(1,i_rd):lims_r(2,i_rd),&
                            &lims_c(1,i_cd):lims_c(2,i_cd)) = mat_dis(&
                            &limsl_r(1):limsl_r(2),limsl_c(1):limsl_c(2))
                    end if
                end do
            end if
        end do

        ! add them together
        call dgsum2d(ctxt,'all',' ',size(mat_loc,1),size(mat_loc,2),mat_loc,&
            &size(mat_loc,1),proc_dest(1),proc_dest(2))
    end function mat_dis2loc

    !> Copies  vacuum  variables  of  a previous  level  into  the  appropriate,
    !! interlaced place of a next level.
    !!
    !! This is done  by multiplying left by  \f$\overline{\text{T}}\f$ and right
    !! by \f$\overline{\text{T}}^T\f$ with
    !! \f[
    !!  a_{ij} =
    !!      \left\{\begin{aligned}
    !!          1 \quad &\text{for}  \left(i,j\right) =
    !!              \left(2j-1+(i-1)n_\text{old},j+(i-1)n_\text{old}\right) \\
    !!          0 \quad &\text{otherwise}
    !!      \end{aligned}\right.
    !! \f]
    !! where \f$i\f$ ranges from \f$1\f$  to \f$n_\text{old}\f$ and \f$j\f$ from
    !! \f$1\f$ to \f$m_\text{old}\f$, with the size of \f$\overline{\text{T}}\f$
    !! equal to \f$n_\text{new} \times n_\text{old}  \f$ where \f$n\f$ refers to
    !! \c n_bnd(1) and \f$m\f$ to \c n_bnd(2).
    integer function interlaced_vac_copy(vac_old,vac) result(ierr)
#if ldebug
        use num_vars, only: ltest, n_procs, rank
        use input_utilities, only: get_log
#endif

        character(*), parameter :: rout_name = 'interlaced_vac_copy'

        ! input / output
        type(vac_type), intent(in) :: vac_old                                   !< old vacuum
        type(vac_type), intent(inout) :: vac                                    !< vacuum

        ! local variables
        integer :: id                                                           ! counter
        integer :: i_cd                                                         ! index of subcol
        integer :: i_rd                                                         ! index of subrow
        integer :: rd, cd                                                       ! global counter for row and col
        integer :: rdl, cdl                                                     ! local counter for row and col
        integer :: alpha_id                                                     ! field line on which an index is situated
        integer :: par_id                                                       ! index point on field line
        integer :: n_ang(2)                                                     ! number of points in angular directions
        integer :: n_ang_old(2)                                                 ! number of points in angular directions in old vacuum
        real(dp), allocatable :: loc_a(:,:)                                     ! local transformation matrix
        real(dp), allocatable :: loc_b(:,:)                                     ! local dummy matrix
        integer :: desc_loc(blacsctxtsize)                                      ! descriptor for loc_A and loc_B
        integer :: desc_loc_old(blacsctxtsize)                                  ! descriptor for loc_A and loc_B in old vacuum context
        character(len=max_str_ln) :: err_msg                                    ! error message
#if ldebug
        logical :: test_redist                                                  ! whether to test the redistribution of H and G
        real(dp), allocatable :: hg_ser(:,:)                                    ! serial versions of H and G
        real(dp), allocatable :: hg_ser_old(:,:)                                ! serial versions of H and G in old vacuum
#endif

        ! initialize ierr
        ierr = 0

```

```
        ! initialize
        n_ang = vac%n_ang
        n_ang_old = vac_old%n_ang

        ! test
        if (n_ang(2).ne.n_ang_old(2)) then
            ierr = 1
            err_msg = 'Old and new vacuum need to have an equal number of &
                &field lines'
            chckerr(err_msg)
        end if
        if (n_ang(1).ne.2*n_ang_old(1)-1) then
            ierr = 1
            err_msg = 'Old and new vacuum need to have a compatible number of &
                &points along the field lines'
            chckerr(err_msg)
        end if

        ! loop over all field lines
        do id = 1,n_ang(2)
            ! copy normal and position vector
            vac%norm((id-1)*n_ang(1)+1:id*n_ang(1):2,:) = &
                &vac_old%norm((id-1)*n_ang_old(1)+1:id*n_ang_old(1):1,:)
            vac%x_vec((id-1)*n_ang(1)+1:id*n_ang(1):2,:) = &
                &vac_old%x_vec((id-1)*n_ang_old(1)+1:id*n_ang_old(1):1,:)

            ! copy variables specific to style
            select case (vac%style)
                case (1)                                                    ! field-line 3-D
                    vac%h_fac((id-1)*n_ang(1)+1:id*n_ang(1):2,:) = vac_old%&
                        &h_fac((id-1)*n_ang_old(1)+1:id*n_ang_old(1):1,:)
                case (2)                                                    ! axisymmetric
                    vac%dnorm((id-1)*n_ang(1)+1:id*n_ang(1):2,:) = &
                        &vac_old%dnorm((id-1)*n_ang_old(1)+1:id*n_ang_old(1):1,:)
                    vac%ang((id-1)*n_ang(1)+1:id*n_ang(1):2,:) = &
                        &vac_old%ang((id-1)*n_ang_old(1)+1:id*n_ang_old(1):1,:)
            end select
        end do

        ! set up transformation matrix A and dummy matrix B
        allocate(loc_a(vac%n_loc(1),vac_old%n_loc(2)))
        allocate(loc_b(vac%n_loc(1),vac_old%n_loc(2)))
        loc_a = 0._dp
        loc_b = 0._dp
        subcols: do i_cd = 1,size(vac_old%lims_c,2)
            col: do cd = vac_old%lims_c(1,i_cd),vac_old%lims_c(2,i_cd)
                ! set field line index and point index and calculate row
                alpha_id = (cd-1)/n_ang_old(1)+1
                par_id = mod(cd-1,n_ang_old(1))+1
                rd = 2*par_id-1 + (alpha_id-1)*n_ang(1)

                ! set local column index
                cdl = sum(vac_old%lims_c(2,1:i_cd-1)-&
                    &vac_old%lims_c(1,1:i_cd-1)+1) + &
                    &cd-vac_old%lims_c(1,i_cd)+1

                subrows: do i_rd = 1,size(vac%lims_r,2)
                    ! set local row index if in this subrow
                    if (rd.ge.vac%lims_r(1,i_rd) .and. &
                        &rd.le.vac%lims_r(2,i_rd)) then
                        rdl = sum(vac%lims_r(2,1:i_rd-1)-&
                            &vac%lims_r(1,1:i_rd-1)+1) + &
                            &rd-vac%lims_r(1,i_rd)+1

                        loc_a(rdl,cdl) = 1._dp
                    end if
                end do subrows
            end do col
        end do subcols
        call descinit(desc_loc,vac%n_bnd,vac_old%n_bnd,vac%bs,&
            &vac%bs,0,0,vac%ctxt_HG,max(1,vac%n_loc(1)),ierr)
        chckerr('descinit failed for loc')
        call descinit(desc_loc_old,vac%n_bnd,vac_old%n_bnd,vac%bs,&
            &vac%bs,0,0,vac_old%ctxt_HG,max(1,vac%n_loc(1)),ierr)
        chckerr('descinit failed for loc_old')

        ! treat H
        chckerr('')
        call pdgemm('N','N',vac%n_bnd,vac_old%n_bnd,vac_old%n_bnd,&
            &1._dp,loc_a,1,1,desc_loc_old,vac_old%H,1,1,&
            &vac_old%desc_H,0._dp,loc_b,1,1,desc_loc_old)
        call pdgemm('N','T',vac%n_bnd,vac%n_bnd,vac_old%n_bnd,&
            &1._dp,loc_b,1,1,desc_loc,loc_a,1,1,&
            &desc_loc,0._dp,vac%H,1,1,vac%desc_H)

        ! treat G
        call pdgemm('N','N',vac%n_bnd,vac_old%n_bnd,vac_old%n_bnd,&
```

```
            &1._dp,loc_a,1,1,desc_loc_old,vac_old%G,1,1,&
            &vac_old%desc_G,0._dp,loc_b,1,1,desc_loc_old)
        call pdgemm('N','T',vac%n_bnd,vac%n_bnd,vac_old%n_bnd,&
            &1._dp,loc_b,1,1,desc_loc,loc_a,1,1,&
            &desc_loc,0._dp,vac%G,1,1,vac%desc_G)

#if ldebug
        ! test
        if (ltest) then
            call writo('test resdistribution of H?')
            test_redist = get_log(.false.)
        else
            test_redist = .false.
        end if
        if (test_redist) then
            allocate(hg_ser(vac%n_bnd,vac%n_bnd))
            allocate(hg_ser_old(vac_old%n_bnd,vac_old%n_bnd))
            ierr = mat_dis2loc(vac_old%ctxt_HG,vac_old%H,&
                &vac_old%lims_r,vac_old%lims_c,hg_ser_old,&
                &proc=n_procs-1)
            chckerr('')
            ierr = mat_dis2loc(vac%ctxt_HG,vac%H,&
                &vac%lims_r,vac%lims_c,hg_ser,&
                &proc=n_procs-1)
            chckerr('')
            if (rank.eq.n_procs-1) then
                call writo('old H:',persistent=rank.eq.n_procs-1)
                call print_ar_2(hg_ser_old)
                call writo('new H:',persistent=rank.eq.n_procs-1)
                call print_ar_2(hg_ser)
                call plot_hdf5('HG_ser_old','HG_ser_old',&
                    &reshape(hg_ser_old,[vac_old%n_bnd,vac_old%n_bnd,1]))
                call plot_hdf5('HG_ser','HG_ser',&
                    &reshape(hg_ser,[vac%n_bnd,vac%n_bnd,1]))
            end if
        end if
#endif

        ! clean up
        deallocate(loc_a,loc_b)
    end function interlaced_vac_copy
end module vac_utilities
```

D.90 /Users/toonweyens/Code/PB3D/Modules/vac_vars.f90 File Reference

Interfaces and Types

- type vac_vars::vac_type

  vacuum type

Modules

- module vac_vars

  Variables pertaining to the vacuum quantities.

Functions/Subroutines

- integer function vac_vars::init_vac (vac, style, n_bnd, prim_x, n_ang, jq)

  Initializes a vacuum type.
- subroutine, public vac_vars::set_loc_lims (n_loc, bs, ind_p, n_p, lims)

  Calculates the limits in local index.
- subroutine vac_vars::dealloc_vac (vac)

  Deallocates vacuum variables.
- integer function, public vac_vars::copy_vac (vac, vac_copy)

  Copy a vacuum type.
- logical function, public vac_vars::in_context (ctxt)

  Indicates whether current process is in the context.

Variables

- integer, public vac_vars::n_alloc_vacs

  nr. of allocated vacs

D.91 vac_vars.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------!
!> Variables pertaining to the vacuum quantities.
!-----------------------------------------------------------------------!
module vac_vars
#include <PB3D_macros.h>
    use strumpackdensepackage
    use str_utilities
    use messages
    use num_vars, only: dp, max_name_ln, iu, weight_dp, max_str_ln
    use grid_vars, only: grid_type

    implicit none

    private
    public copy_vac, set_loc_lims, in_context
#if ldebug
    public n_alloc_vacs
#endif

    ! global variables
    integer, parameter :: bs = 16                                   !< Blocksize of the 2D block-cyclic distribution
#if ldebug
    integer :: n_alloc_vacs                                         !< nr. of allocated vacs \ldebug
#endif

    !> vacuum type
    !!
    !! The arrays here are of the form:
    !!  - \c H, \c G:   <tt>(n_loc,n_loc)</tt>
    !!  - \c res:       <tt>(n_mod_X,n_mod_X)</tt>
    !!
    !! where \c n_loc is the number of  points in the boundary, i.e. a subset of
    !! \c n_bnd.
    !!
    !! For  vacuum style  1, the  vacuum  is assumed  to be  equidsitant in  the
    !! coordinate  along the  magnetic field  lines and,  if there  are multiple
    !! field  lines, also  in  the  field line  label  \f$\alpha\f$. The  limits
    !! on  the  angles are  assumed  to  be given  by  the  global variables  \c
    !! grid_vars.min_par_x, \c  grid_vars.max_par_x and  \c grid_vars.min_alpha,
    !! \c grid_vars.max_alpha.
    !!
    !! For vacuum style 2,  there is an additional angle \c  ang. It is composed
    !! of the  angles along the magnetic  fields (which refers to  \c angle_1 in
    !! the  discussion  of  \c  grid_vars.grid_type),  of  which  there  can  be
    !! multiple, but the total sum must be equal to \c n_bnd.
    type, public :: vac_type
        integer :: style                                            !< style of vacuum (1: field-line 3-D, 2: axisymmetric)
        integer :: prim_x                                           !< primary mode number
        integer :: ctxt_hg                                          !< context for H and G
        integer :: n_bnd                                            !< number of points in boundary
        integer :: bs                                               !< block size in cyclical storage
        integer :: mpi_comm                                         !< communicator for vacuum
        integer :: desc_h(blacsctxtsize)                            !< descriptor for H
        integer :: desc_g(blacsctxtsize)                            !< descriptor for G
        integer :: n_p(2)                                           !< nr. of processes in grid
        integer :: n_ang(2)                                         !< number of angles (1) and number of field lines (2)
        integer :: ind_p(2)                                         !< index of local process in grid
        integer :: n_loc(2)                                         !< local number of rows and columns
        integer, allocatable :: sec_x(:)                            !< secondary mode numbers
        integer, allocatable :: lims_c(:,:)                         !< column limits for different subrows of G and H
        integer, allocatable :: lims_r(:,:)                         !< row limits for different subcolumns of G and H
        real(dp) :: jq                                              !< iota (tor. flux) or q (pol. flux) at edge
        real(dp), allocatable :: ang(:,:)                           !< angle along field line, for each field line
        real(dp), allocatable :: norm(:,:)                          !< J nabla psi normal vector
```

```
        real(dp), allocatable :: dnorm(:,:)                                 !< poloidal derivative of norm (only for style 2)
        real(dp), allocatable :: h_fac(:,:)                                 !< metric factors (1,1), (1,3) and (3,3) (only for style 1)
        real(dp), allocatable :: x_vec(:,:)                                 !< Cartesian vector of position
        real(dp), allocatable :: h(:,:)                                     !< H coefficient
        real(dp), allocatable :: g(:,:)                                     !< G coefficient
        complex(dp), allocatable :: res(:,:)                                !< vacuum response
#if ldebug
        real(dp) :: estim_mem_usage                                         !< estimated memory usage \ldebug
#endif
    contains
        !> initialize
        procedure :: init => init_vac
        !> deallocate
        procedure :: dealloc => dealloc_vac
    end type

contains
    !> \public Initializes a vacuum type.
    !!
    !! The number of modes as well as \c n and \c m are also set up.
    !!
    !! The  variables G  and H  are  saved in  a  special format,  based on  the
    !! block-cyclical  distribution employed  in  scalapack.  As a  hypothetical
    !! example, consider a process grid  corresponding to block-size 1x2 and 2x3
    !! processes with a total size of 3x15:
    !! \f[\left[\begin{array}{cccccccccccccc}
    !!  0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 \\
    !!  3 & 3 & 4 & 4 & 5 & 5 & 3 & 3 & 4 & 4 & 5 & 5 & 3 & 3 & 4 \\
    !!  0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1
    !! \end{array}\right]\f]
    !!
    !! Therefore, the 3-D storage convention used is, for example, for process 1:
    !! \f[\begin{bmatrix}
    !!  (1,3) & (1,4) & (1,9) & (1,10) & (1,15) & (1,\cdot) \\
    !!  (3,3) & (3,4) & (3,9) & (3,10) & (3,15) & (3,\cdot)
    !! \end{bmatrix}\f]
    !!
    !! The  information  concerning  the  delimitations  of  the  individual
    !! subintervals in the  horizontal and vertical direction can  be saved once
    !! per subinterval.
    !! For process 4 of above example, this would be
    !! \f[
    !!  \begin{bmatrix} 2 \\ 2 \end{bmatrix} ;
    !! \f]
    !! for the vertical direction, and
    !! \f[
    !!  \begin{bmatrix} 3 & 9 & 15 \\ 4 & 10 & 15 \end{bmatrix} ;
    !! \f]
    !! for the horizontal direction.
    !!
    !! A value  \f$\cdot\f$ indicates that  the index  is out of  global bounds.
    !! Practically, this is  checked by seeing whether the total  index does not
    !! exceed the global bounds.
    !!
    !! \return ierr
    integer function init_vac(vac,style,n_bnd,prim_X,n_ang,jq) result(ierr)
        use mpi
        use num_vars, only: n_procs, rank
        use x_vars, only: n_mod_x
#if ldebug
        use num_vars, only: print_mem_usage
#endif

        character(*), parameter :: rout_name = 'init_vac'

        ! input / output
        class(vac_type), intent(inout) :: vac                               !< vacuum variables
        integer, intent(in) :: style                                        !< style of vacuum (1: field-line 3-D, 2: axisymmetric)
        integer, intent(in) :: n_bnd                                        !< total number of points in boundary
        integer, intent(in) :: prim_x                                       !< primary mode number
        integer, intent(in) :: n_ang(2)                                     !< number of angles (1) and number of field lines (2)
        real(dp), intent(in) :: jq                                          !< iota (tor. flux) or q (pol. flux) at edge

        ! local variables
        character(len=max_str_ln) :: err_msg                                ! error message
        integer :: n_dims                                                   ! number of dimensions to be saved in x_vec and norm
        integer :: jd                                                       ! counter
        integer, allocatable :: proc_map(:,:)                               ! process map

        ! initialize ierr
        ierr = 0

```

```
#if ldebug
        ! initialize memory usage
        if (print_mem_usage) vac%estim_mem_usage = 0._dp
#endif

        ! test
        if (product(n_ang).ne.n_bnd) then
            ierr = 1
            err_msg = 'The total number of angles along B, '//&
                &trim(i2str(product(n_ang)))//', is not equal to n_bnd = '//&
                &trim(i2str(n_bnd))
            chckerr(err_msg)
        end if

        ! set some variables
        vac%bs = bs
        if (n_procs.eq.1) vac%bs = n_bnd
        vac%jq = jq
        vac%n_ang = n_ang

        ! Initialize the BLACS grid for H and G
        vac%prim_X = prim_x
        vac%n_bnd = n_bnd
        vac%n_p(1)=floor(sqrt(n_procs*1._dp))
        vac%n_p(2)=n_procs/vac%n_p(1)
        allocate(proc_map(vac%n_p(1),vac%n_p(2)))
        proc_map = transpose(reshape(&
            &[(jd+n_procs-1-product(vac%n_p),jd=1,product(vac%n_p))],vac%n_p))
        call blacs_get(0,0,vac%ctxt_HG)
        call blacs_gridmap(vac%ctxt_HG,proc_map,vac%n_p(1),vac%n_p(1),&
            &vac%n_p(2))                                                ! last MPI proc is also last process in map
        call blacs_gridinfo(vac%ctxt_HG,vac%n_p(1),vac%n_p(2),vac%ind_p(1),&
            &vac%ind_p(2))                                              ! get nr. of rows and columns and local index
        if(minval(vac%ind_p).ge.0) then                                 ! only the ranks that participate
            do jd = 1,2                                                 ! loop over rows, then columns
                vac%n_loc(jd) = numroc(vac%n_bnd,vac%bs,vac%ind_p(jd),0,&
                    &vac%n_p(jd))                                       ! get local number of rows, then columns
            end do
            call descinit(vac%desc_H,n_bnd,n_bnd,vac%bs,vac%bs,0,0,vac%ctxt_HG,&
                &max(1,vac%n_loc(1)),ierr)
            chckerr('descinit failed')
            call descinit(vac%desc_G,n_bnd,n_bnd,vac%bs,vac%bs,0,0,vac%ctxt_HG,&
                &max(1,vac%n_loc(1)),ierr)
            chckerr('descinit failed')
            call mpi_comm_split(mpi_comm_world,1,rank,vac%MPI_Comm,ierr)
            chckerr('')
        else
            vac%n_loc = [0,0]
            call mpi_comm_split(mpi_comm_world,mpi_undefined,rank,&
                &vac%MPI_Comm,ierr)
            chckerr('')
        end if

        ! set limits
        call set_loc_lims(vac%n_loc(1),vac%bs,vac%ind_p(1),vac%n_p(1),&
            &vac%lims_r)
        call set_loc_lims(vac%n_loc(2),vac%bs,vac%ind_p(2),vac%n_p(2),&
            &vac%lims_c)

        ! set n_dims
        vac%style = style
        select case (vac%style)
            case (1)                                                    ! field-line 3-D
                n_dims = 3
            case (2)                                                    ! axisymmetric
                n_dims = 2
            case default
                ierr = 1
                err_msg = 'No vacuum style '//trim(i2str(vac%style))//&
                    &' possible'
                chckerr(err_msg)
        end select

        ! allocate normal and position vector
        allocate(vac%norm(vac%n_bnd,n_dims))
        allocate(vac%x_vec(vac%n_bnd,n_dims))

        ! allocate variables specific to style
        select case (vac%style)
            case (1)                                                    ! field-line 3-D
                allocate(vac%h_fac(vac%n_bnd,4))
            case (2)                                                    ! axisymmetric
                allocate(vac%ang(n_ang(1),n_ang(2)))
                allocate(vac%dnorm(vac%n_bnd,n_dims))
        end select

```

```
        ! allocate vacuum response if last process
        if (rank.eq.n_procs-1) allocate(vac%res(n_mod_x,n_mod_x))

        ! allocate H and G
        allocate(vac%G(vac%n_loc(1),vac%n_loc(2)))
        allocate(vac%H(vac%n_loc(1),vac%n_loc(2)))
        vac%G = 0._dp
        vac%H = 0._dp

#if ldebug
        ! set estimated memory usage
        if (print_mem_usage) vac%estim_mem_usage = &
            &vac%estim_mem_usage + size(vac%G) + n_mod_x**2

        ! increment n_alloc_vacs
        n_alloc_vacs = n_alloc_vacs + 1

        ! print memory usage
        if (print_mem_usage) call writo('[rank '//trim(i2str(rank))//&
            &' - Expected memory usage of vac: '//&
            &trim(r2strt(vac%estim_mem_usage*weight_dp*2))//' kB]',alert=.true.)
#endif
    end function init_vac

    !> Calculates the limits in local index.
    !!
    !! \see See init_var() for an explanation.
    subroutine set_loc_lims(n_loc,bs,ind_p,n_p,lims)
        ! input / output
        integer, intent(in) :: n_loc                                    !< number of points owned by local process
        integer, intent(in) :: bs                                       !< block size
        integer, intent(in) :: ind_p                                    !< index of current process in this dimension
        integer, intent(in) :: n_p                                      !< number of processes in this dimension
        integer, intent(inout), allocatable :: lims(:,:)                !< limits for different subregions in this dimension

        ! local variables
        integer :: jd                                                   ! counter

        ! set limits
        if (n_loc.gt.0) then
            allocate(lims(2,ceiling(n_loc*1._dp/bs)))
            do jd = 1,size(lims,2)
                lims(1,jd) = indxl2g((jd-1)*bs+1,bs,ind_p,0,n_p)
                lims(2,jd) = indxl2g(min(jd*bs,n_loc),bs,ind_p,0,n_p)
            end do
        else
            ! these processes don't play along
            allocate(lims(2,1))
            lims(:,1) = [0,-1]
        end if
    end subroutine set_loc_lims

    !> \public Deallocates vacuum variables.
    subroutine dealloc_vac(vac)
#if ldebug
        use num_vars, only: rank, print_mem_usage
#endif

        ! input / output
        class(vac_type), intent(inout) :: vac                           !< vacuum variables to be deallocated

#if ldebug
        ! local variables
        integer :: mem_diff                                             ! difference in memory
        real(dp) :: estim_mem_usage                                     ! estimated memory usage

        ! memory usage before deallocation
        if (print_mem_usage) then
            mem_diff = get_mem_usage()
            estim_mem_usage = vac%estim_mem_usage
        end if
#endif

        ! free grid
        if (in_context(vac%ctxt_HG)) call blacs_gridexit(vac%ctxt_HG)

        ! deallocate allocatable variables
        call dealloc_vac_final(vac)

#if ldebug
        ! decrement n_alloc_vacs
        n_alloc_vacs = n_alloc_vacs - 1

        ! memory usage difference after deallocation
        if (print_mem_usage) then
```

```
            mem_diff = mem_diff - get_mem_usage()
            call writo('[Rank '//trim(i2str(rank))//' - liberated '//&
                &trim(i2str(mem_diff))//'kB deallocating vac ('//&
                &trim(i2str(nint(100*mem_diff/&
                &(estim_mem_usage*weight_dp*2))))//&
                &'% of estimated)]',alert=.true.)
        end if
#endif
    contains
        ! Note: intent(out) automatically deallocates the variable
        !> \private
        subroutine dealloc_vac_final(vac)
            ! input / output
            type(vac_type), intent(out) :: vac                              ! vacuum variables to be deallocated
        end subroutine dealloc_vac_final
    end subroutine dealloc_vac

    !> Copy a vacuum type.
    !!
    !! \note The copy should be unallocated.
    integer function copy_vac(vac,vac_copy) result(ierr)
        use num_vars, only: rank, n_procs

        ! input / output
        class(vac_type), intent(in) :: vac                                  !< vac to be copied
        class(vac_type), intent(inout) :: vac_copy                          !< copy

        character(*), parameter :: rout_name = 'copy_vac'

        ! initialize ierr
        ierr = 0

        ! reallocate
        ierr = vac_copy%init(vac%style,vac%n_bnd,vac%prim_X,vac%n_ang,&
            &vac%jq)
        chckerr('')

        ! copy normal and position vector
        vac_copy%norm = vac%norm
        vac_copy%x_vec = vac%x_vec

        ! copy variables specific to style
        select case (vac%style)
            case (1)                                                        ! field-line 3-D
                vac_copy%h_fac = vac%h_fac
            case (2)                                                        ! axisymmetric
                vac_copy%dnorm = vac%dnorm
                vac_copy%ang = vac%ang
        end select

        ! copy vacuum response
        if (rank.eq.n_procs-1) vac_copy%res = vac%res

        ! copy H and G
        vac_copy%H = vac%H
        vac_copy%G = vac%G
    end function copy_vac

    !> Indicates whether current process is in the context.
    logical function in_context(ctxt) result(res)
        ! input / output
        integer, intent(in) :: ctxt                                         !< context for vector

        ! local variables
        integer :: n_p(2)                                                   ! nr. of processes in grid
        integer :: ind_p(2)                                                 ! index of local process in grid

        call blacs_gridinfo(ctxt,n_p(1),n_p(2),ind_p(1),ind_p(2))
        res = ind_p(1).ge.0
    end function in_context
end module vac_vars
```

D.92 /Users/toonweyens/Code/PB3D/Modules/VMEC_ops.f90 File Reference

Modules

- module vmec_ops

  Operations that concern the output of VMEC.

Functions/Subroutines

- integer function, public vmec_ops::read_vmec (n_r_in, use_pol_flux_v)

  Reads the VMEC equilibrium data.
- subroutine, public vmec_ops::normalize_vmec

  Normalizes VMEC input.

D.93 VMEC_ops.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Operations that concern the output of VMEC.
!-----------------------------------------------------------------------------!
module vmec_ops
#include <PB3D_macros.h>
    use str_utilities
    use output_ops
    use messages
    use num_vars, only: &
        &dp, max_str_ln, pi
    use read_wout_mod, only: read_wout_file, read_wout_deallocate, &            ! from LIBSTELL
        &n_r_vmec => ns, &                                                      ! n_r
        &xn, xm, &                                                              ! xn, xm
        &lrfp, &                                                                ! whether or not the poloidal flux is used as radial variable
        &phi, phipf, &                                                          ! toroidal flux (FM), norm. deriv. of toroidal flux (FM)
        &iotaf, &                                                               ! rot. transf. (tor. flux) or saf. fac. (pol. flux) (FM)
        &presf, &                                                               ! pressure (FM)
        &bsubumns, bsubumnc, bsubvmns, bsubvmnc, bsubsmns, bsubsmnc, &          ! B_theta (HM), B_zeta (HM), B_r (FM)
        &jcuru, jcurv, &                                                        ! poloidal (FM) and toroidal (FM) current
        &bmns, bmnc, &                                                          ! magnitude of B (HM)
        &lmns, lmnc, rmns, rmnc, zmns, zmnc, &                                  ! lambda (HM), R (FM), Z(FM)
        &gmnc, gmns                                                             ! Jacobian in VMEC coordinates
    use vmec_vars

    implicit none
    private
    public read_vmec, normalize_vmec

contains
    !> Reads the VMEC equilibrium data.
    !!
    !! \return ierr
    integer function read_vmec(n_r_in,use_pol_flux_V) result(ierr)
        use num_utilities, only: calc_int, spline
        use num_vars, only: eq_name, max_deriv, norm_disc_prec_eq

        character(*), parameter :: rout_name = 'read_VMEC'

        ! input / output
        integer, intent(inout) :: n_r_in                                        !< nr. of normal points in input grid
        logical, intent(inout) :: use_pol_flux_v                                !< .true. if VMEC equilibrium is based on poloidal flux

        ! local variables
        integer :: id, kd                                                       ! counters
        real(dp), allocatable :: r_v(:)                                         ! normal coordinate
        real(dp), allocatable :: l_c_h(:,:,:)                                   ! temporary HM variable
        real(dp), allocatable :: l_s_h(:,:,:)                                   ! temporary HM variable
        real(dp), allocatable :: jac_c_h(:,:,:)                                 ! temporary HM variable
        real(dp), allocatable :: jac_s_h(:,:,:)                                 ! temporary HM variable
        character(len=max_str_ln) :: err_msg                                    ! error message
        character(len=8) :: flux_name                                           ! either poloidal or toroidal
        real(dp), allocatable :: b_v_sub_c_m(:,:,:), b_v_sub_s_m(:,:,:)         ! Coeff. of B_i in (co)sine series (r,theta,phi) (FM, HM, HM)
#if ldebug
        real(dp), allocatable :: b_v_c_h(:,:), b_v_s_h(:,:)                     ! Coeff. of magnitude of B (HM)
#endif

        ! initialize ierr
        ierr = 0

        call writo('Reading data from VMEC output "' &
            &// trim(eq_name) // '"')
```

```
        call lvl_ud(1)

        ! read VMEC output using LIBSTELL
        if (scan(eq_name, ".") .ne. scan(eq_name, ".", .true.)) &
            &call writo('There seems to be a dot "."  in "'//trim(eq_name)//&
            &'". This can create problems.', alert=.true.)
        call read_wout_file(eq_name,ierr)                                   ! read the VMEC file
        chckerr('Failed to read the VMEC file')

        ! set some variables
        n_r_in = n_r_vmec
        allocate(flux_t_v(n_r_in,0:max_deriv+1))
        allocate(flux_p_v(n_r_in,0:max_deriv+1))
        allocate(rot_t_v(n_r_in,0:max_deriv+1))
        allocate(q_saf_v(n_r_in,0:max_deriv+1))
        allocate(pres_v(n_r_in,0:max_deriv+1))
        use_pol_flux_v = lrfp
        flux_t_v(:,0) = phi
        flux_t_v(:,1) = phipf
        flux_p_v(:,1) = iotaf*phipf
        ierr = calc_int(flux_p_v(:,1),1.0_dp/(n_r_in-1.0_dp),flux_p_v(:,0))
        chckerr('')
        pres_v(:,0) = presf
        rot_t_v(:,0) = iotaf
        q_saf_v(:,0) = 1._dp/iotaf
        if (use_pol_flux_v) then
            flux_name = 'poloidal'
        else
            flux_name = 'toroidal'
        end if
        b_0_v = sum(1.5*bmnc(:,2)-0.5_dp*bmnc(:,3))                         ! convert to full mesh

        call writo('VMEC version is ' // trim(r2str(vmec_version)))
        if (is_freeb_v) then
            call writo('Free boundary VMEC')
        else
            call writo('Fixed boundary VMEC')
        end if
        if (is_asym_v) then
            call writo('No stellarator symmetry')
        else
            call writo('Stellarator symmetry')
        end if
        call writo('VMEC has '//trim(i2str(mpol_v))//' poloidal and '&
            &//trim(i2str(ntor_v))//' toroidal modes,')
        if (ntor_v.gt.0) call writo('(basic toroidal field period NFP = '//&
            &trim(i2str(nfp_v))//')')
        call writo('defined on '//trim(i2str(n_r_in))//' '//flux_name//&
            &' flux surfaces')

        call writo('Running tests')
        call lvl_ud(1)
        if (mnmax_v.ne.((ntor_v+1)+(2*ntor_v+1)*(mpol_v-1))) then           ! for mpol_V = 0, only ntor_V+1 values needed
            err_msg = 'Inconsistency in ntor_V, mpol_V and mnmax_V'
            ierr = 1
            chckerr(err_msg)
        end if
        call lvl_ud(-1)

        call writo('Updating variables')
        allocate(mn_v(mnmax_v,2))
        mn_v(:,1) = nint(xm(1:mnmax_v))
        mn_v(:,2) = nint(xn(1:mnmax_v))

        call lvl_ud(-1)
        call writo('Data from VMEC output successfully read')

        call writo('Saving VMEC output to PB3D')
        call lvl_ud(1)

        ! Allocate the Fourier coefficients
        allocate(r_v_c(mnmax_v,n_r_in,0:max_deriv+1))
        allocate(r_v_s(mnmax_v,n_r_in,0:max_deriv+1))
        allocate(z_v_c(mnmax_v,n_r_in,0:max_deriv+1))
        allocate(z_v_s(mnmax_v,n_r_in,0:max_deriv+1))
        allocate(l_v_c(mnmax_v,n_r_in,0:max_deriv+1))
        allocate(l_v_s(mnmax_v,n_r_in,0:max_deriv+1))
        allocate(jac_v_c(mnmax_v,n_r_in,0:max_deriv))
        allocate(jac_v_s(mnmax_v,n_r_in,0:max_deriv))
        allocate(l_c_h(mnmax_v,n_r_in,0:0))
        allocate(l_s_h(mnmax_v,n_r_in,0:0))
        allocate(jac_c_h(mnmax_v,n_r_in,0:0))
        allocate(jac_s_h(mnmax_v,n_r_in,0:0))

        ! factors R_V_c,s; Z_V_c,s and L_C,s and HM varieties
        r_v_c(:,:,0) = rmnc(1:mnmax_v,:)
```

```
        z_v_s(:,:,0) = zmns(1:mnmax_v,:)
        l_s_h(:,:,0) = lmns(1:mnmax_v,:)
        jac_c_h(:,:,0) = gmnc(1:mnmax_v,:)
        if (is_asym_v) then                                          ! following only needed in asymmetric situations
            r_v_s(:,:,0) = rmns(1:mnmax_v,:)
            z_v_c(:,:,0) = zmnc(1:mnmax_v,:)
            l_c_h(:,:,0) = lmnc(1:mnmax_v,:)
            jac_s_h(:,:,0) = gmns(1:mnmax_v,:)
        else
            r_v_s = 0._dp
            z_v_c = 0._dp
            l_c_h = 0._dp
            jac_s_h = 0._dp
        end if

        ! calculate data for normal derivatives  with the toroidal (or poloidal)
        ! flux, normalized wrt. to the  maximum flux, equidistantly, so the step
        ! size is 1/(n_r_in-1).
        allocate(r_v(n_r_in))
        r_v = [((kd-1._dp)/(n_r_in-1),kd=1,n_r_in)]

        ! up to order 2, full mesh
        do kd = 1,2
            ierr = spline(r_v,q_saf_v(:,0),r_v,q_saf_v(:,kd),&
                &ord=norm_disc_prec_eq,deriv=kd)
            chckerr('')
            ierr = spline(r_v,rot_t_v(:,0),r_v,rot_t_v(:,kd),&
                &ord=norm_disc_prec_eq,deriv=kd)
            chckerr('')
            ierr = spline(r_v,pres_v(:,0),r_v,pres_v(:,kd),&
                &ord=norm_disc_prec_eq,deriv=kd)
            chckerr('')
            ierr = spline(r_v,flux_t_v(:,0),r_v,flux_t_v(:,kd),&
                &ord=norm_disc_prec_eq,deriv=kd)
            chckerr('')
            ierr = spline(r_v,flux_p_v(:,0),r_v,flux_p_v(:,kd),&
                &ord=norm_disc_prec_eq,deriv=kd)
            chckerr('')
        end do

        ! up to order 2, half mesh: also extrapolate
        do kd = 0,2
            do id = 1,mnmax_v
                ierr = spline(-0.5_dp/n_r_in+r_v(2:n_r_in),&
                    &jac_c_h(id,2:n_r_in,0),r_v,jac_v_c(id,:,kd),&
                    &ord=norm_disc_prec_eq,deriv=kd,extrap=.true.)
                chckerr('')
                if (is_asym_v) then
                    ierr = spline(-0.5_dp/n_r_in+r_v(2:n_r_in),&
                        &jac_s_h(id,2:n_r_in,0),r_v,jac_v_s(id,:,kd),&
                        &ord=norm_disc_prec_eq,deriv=kd,extrap=.true.)
                    chckerr('')
                end if
            end do
        end do

        ! up to order 2, full mesh
        do kd = 1,3
            do id = 1,mnmax_v
                ! full mesh
                ierr = spline(r_v,r_v_c(id,:,0),r_v,r_v_c(id,:,kd),&
                    &ord=norm_disc_prec_eq,deriv=kd)
                chckerr('')
                ierr = spline(r_v,z_v_s(id,:,0),r_v,z_v_s(id,:,kd),&
                    &ord=norm_disc_prec_eq,deriv=kd)
                chckerr('')
                if (is_asym_v) then
                    ierr = spline(r_v,r_v_s(id,:,0),r_v,r_v_s(id,:,kd),&
                        &ord=norm_disc_prec_eq,deriv=kd)
                    chckerr('')
                    ierr = spline(r_v,z_v_c(id,:,0),r_v,z_v_c(id,:,kd),&
                        &ord=norm_disc_prec_eq,deriv=kd)
                    chckerr('')
                end if
            end do
        end do

        ! up to order 2, half mesh: also extrapolate
        do kd = 0,3
            do id = 1,mnmax_v
                ierr = spline(-0.5_dp/n_r_in+r_v(2:n_r_in),&
                    &l_s_h(id,2:n_r_in,0),r_v,l_v_s(id,:,kd),&
                    &ord=norm_disc_prec_eq,deriv=kd,extrap=.true.)
                chckerr('')
                if (is_asym_v) then
                    ierr = spline(-0.5_dp/n_r_in+r_v(2:n_r_in),&
```

```
                        &l_c_h(id,2:n_r_in,0),r_v,l_v_c(id,:,kd),&
                        &ord=norm_disc_prec_eq,deriv=kd,extrap=.true.)
                    chckerr('')
                end if
            end do
        end do

        ! These are only necessary if the compiler searches for uninitialized
        ! memory
        flux_t_v(:,3:) = 0._dp
        flux_p_v(:,3:) = 0._dp
        q_saf_v(:,3:) = 0._dp
        rot_t_v(:,3:) = 0._dp
        pres_v(:,3:) = 0._dp

        ! allocate helper variables
        allocate(b_v_sub_c_m(mnmax_v,n_r_in,3)); b_v_sub_c_m = 0._dp
        allocate(b_v_sub_s_m(mnmax_v,n_r_in,3)); b_v_sub_s_m = 0._dp
#if ldebug
        allocate(b_v_c_h(mnmax_v,n_r_in)); b_v_c_h = 0._dp
        allocate(b_v_s_h(mnmax_v,n_r_in)); b_v_s_h = 0._dp
#endif

        ! store in helper variables
        b_v_sub_s_m(:,:,1) = bsubsmns(1:mnmax_v,:)
        b_v_sub_c_m(:,:,2) = bsubumnc(1:mnmax_v,:)
        b_v_sub_c_m(:,:,3) = bsubvmnc(1:mnmax_v,:)
        if (is_asym_v) then                                             ! following only needed in asymmetric situations
            b_v_sub_c_m(:,:,1) = bsubsmnc(1:mnmax_v,:)
            b_v_sub_s_m(:,:,2) = bsubumns(1:mnmax_v,:)
            b_v_sub_s_m(:,:,3) = bsubvmns(1:mnmax_v,:)
        else
            b_v_sub_c_m(:,:,1) = 0._dp
            b_v_sub_s_m(:,:,2) = 0._dp
            b_v_sub_s_m(:,:,3) = 0._dp
        end if
#if ldebug
        b_v_c_h(:,:) = bmnc(1:mnmax_v,:)
        if (is_asym_v) then                                             ! following only needed in asymmetric situations
            b_v_s_h(:,:) = bmns(1:mnmax_v,:)
        else
            b_v_s_h(:,:) = 0._dp
        end if
#endif

        ! allocate FM variables
        allocate(b_v_sub_c(mnmax_v,n_r_in,3))
        allocate(b_v_sub_s(mnmax_v,n_r_in,3))
#if ldebug
        allocate(b_v_c(mnmax_v,n_r_in))
        allocate(b_v_s(mnmax_v,n_r_in))
        allocate(j_v_sup_int(n_r_in,2))
#endif

        ! half mesh: extrapolate
        do id = 1,mnmax_v
            do kd = 2,3
                ierr = spline(-0.5_dp/n_r_in+r_v(2:n_r_in),&
                    &b_v_sub_c_m(id,2:n_r_in,kd),r_v,b_v_sub_c(id,:,kd),&
                    &ord=norm_disc_prec_eq,deriv=0,extrap=.true.)
                chckerr('')
                ierr = spline(-0.5_dp/n_r_in+r_v(2:n_r_in),&
                    &b_v_sub_s_m(id,2:n_r_in,kd),r_v,b_v_sub_s(id,:,kd),&
                    &ord=norm_disc_prec_eq,deriv=0,extrap=.true.)
                chckerr('')
            end do
        end do
#if ldebug
        do id = 1,mnmax_v
            ierr = spline(-0.5_dp/n_r_in+r_v(2:n_r_in),&
                &b_v_c_h(id,2:n_r_in),r_v,b_v_c(id,:),&
                &ord=norm_disc_prec_eq,deriv=0,extrap=.true.)
            chckerr('')
            ierr = spline(-0.5_dp/n_r_in+r_v(2:n_r_in),&
                &b_v_s_h(id,2:n_r_in),r_v,b_v_s(id,:),&
                &ord=norm_disc_prec_eq,deriv=0,extrap=.true.)
            chckerr('')
        end do
        j_v_sup_int(:,1) = jcuru
        j_v_sup_int(:,2) = jcurv
#endif

        ! deallocate repacked variables
        call read_wout_deallocate()

```

```
        call lvl_ud(-1)
        call writo('Conversion complete')
    end function read_vmec

    !> Normalizes VMEC input.
    !!
    !! \note The normal VMEC coordinate runs from 0 to 1, whatever the
    !! normalization.
    subroutine normalize_vmec
        use eq_vars, only: pres_0, psi_0, r_0, b_0

        ! scale the VMEC quantities
        pres_v = pres_v/pres_0
        flux_t_v = flux_t_v/psi_0
        flux_p_v = flux_p_v/psi_0
        r_v_c = r_v_c/r_0
        r_v_s = r_v_s/r_0
        z_v_c = z_v_c/r_0
        z_v_s = z_v_s/r_0
        l_v_c = l_v_c
        l_v_s = l_v_s
        jac_v_c = jac_v_c/(r_0**3)
        jac_v_s = jac_v_s/(r_0**3)
        b_v_sub_s = b_v_sub_s/(r_0*b_0)
        b_v_sub_c = b_v_sub_c/(r_0*b_0)
#if ldebug
        b_v_c = b_v_c/b_0
        b_v_s = b_v_s/b_0
        j_v_sup_int = j_v_sup_int*psi_0/pres_0
#endif
    end subroutine normalize_vmec
end module vmec_ops
```

D.94 /Users/toonweyens/Code/PB3D/Modules/VMEC_utilities.f90 File Reference

Interfaces and Types

- interface vmec_utilities::fourier2real

  Inverse Fourier transformation, from VMEC.

Modules

- module vmec_utilities

  Numerical utilities related to the output of VMEC.

Functions/Subroutines

- integer function, public vmec_utilities::calc_trigon_factors (theta, zeta, trigon_factors)

  Calculate the trigonometric cosine and sine factors.

Variables

- logical, public vmec_utilities::debug_calc_trigon_factors = .false.

  plot debug information for calc_trigon_factors()

D.95 VMEC_utilities.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Numerical utilities related to the output of VMEC.
!-----------------------------------------------------------------------------!
module vmec_utilities
#include <PB3D_macros.h>
    use str_utilities
    use output_ops
    use messages
    use num_vars, only: &
        &dp, max_str_ln, pi
    use vmec_vars

    implicit none
    private
    public calc_trigon_factors, fourier2real
#if ldebug
    public debug_calc_trigon_factors
#endif

    ! global variables
#if ldebug
    logical :: debug_calc_trigon_factors = .false.                          !< plot debug information for calc_trigon_factors() \ldebug
#endif

    ! interfaces

    !> \public Inverse Fourier transformation, from VMEC.
    !!
    !! Also calculates the poloidal or  toroidal derivatives in VMEC coords., as
    !! indicated by the variable \c deriv(2).
    !!
    !! (The normal derivative  is done on the variables in  Fourier space, and should
    !! be provided here in \c varf_i if needed).
    !!
    !! There are two variants:
    !!  -# version using trigon_factors, which is  useful when the grid on which
    !!  the trigonometric  factors are defined  is not regular and  ideally when
    !!  they are reused multiple times.
    !!  -# version using \f$\theta\f$ and  \f$\zeta\f$ directly, which is useful
    !!  for small, unique calculations.
    !!
    !! Both  these  versions  make  use  of  a  factor  that  represents  angular
    !! derivatives. For <tt>deriv = [j,k]</tt>, this is:
    !!  - \f$m^j (-n)^k (-1)^{\frac{j+k+1}{2}}\f$   for \c varf_c,
    !!  - \f$m^j (-n)^k (-1)^{\frac{j+k}{2}}\f$     for \c varf_s,
    !!
    !! where  the  divisions  have  to  be  done  using  integers,  i.e.  without
    !! remainder. The  first two factors  are straightforward, and the  third one
    !! originates in  the change of  sign when deriving a  cosine, but not  for a
    !! sine.
    !!
    !! Finally, depending  on whether  \f$(j+k)\f$ is even  or odd,  the correct
    !! \f$\cos\f$ or \f$\sin\f$ is chosen.
    !!
    !! \return ierr
    interface fourier2real
        !> \public
        module procedure fourier2real_1
        !> \public
        module procedure fourier2real_2
    end interface

contains
    !> \private version with trigonometric factors
    integer function fourier2real_1(varf_c,varf_s,trigon_factors,varr,sym,&
        &deriv) result(ierr)

        character(*), parameter :: rout_name = 'fourier2real_1'

        ! input / output
        real(dp), intent(in) :: varf_c(:,:)                                 !< \f$\cos\f$ factor of variable in Fourier space
        real(dp), intent(in) :: varf_s(:,:)                                 !< \f$\sin\f$ factor of variable in Fourier space
        real(dp), intent(in) :: trigon_factors(:,:,:,:,:)                   !< trigonometric factor cosine and sine at these angles (see calc_trigon_factors())
        real(dp), intent(inout) :: varr(:,:,:)                              !< variable in real space
        logical, intent(in), optional :: sym(2)                             !< whether to use \c varf_c (1) and / or \c varf_s (2)
        integer, intent(in), optional :: deriv(2)                           !< optional derivatives in angular coordinates
```

```fortran

        ! local variables
        character(len=max_str_ln) :: err_msg                                    ! error message
        integer :: dims(3)                                                      ! dimensions of varr
        integer :: id, kd                                                       ! counters
        integer :: deriv_loc(2)                                                 ! local derivative
        logical :: sym_loc(2)                                                   ! local sym
        real(dp) :: deriv_fac                                                   ! factur due to derivatives

        ! initialize ierr
        ierr = 0

        ! set local deriv and sym
        deriv_loc = [0,0]
        sym_loc = [.true.,.true.]
        if (present(deriv)) deriv_loc = deriv
        if (present(sym)) sym_loc = sym

        ! set dimensions
        dims = shape(varr)

        ! test
        if (.not.sym_loc(1) .and. .not.sym_loc(2)) then
            ierr = 1
            err_msg = 'Need at least the cosine or the sine factor'
            chckerr(err_msg)
        end if
        if (size(trigon_factors,1).ne.mnmax_v .and. &
            &size(trigon_factors,2).ne.dims(1) .and. &
            &size(trigon_factors,3).ne.dims(2) .and. &
            &size(trigon_factors,4).ne.dims(3) .and. &
            &size(trigon_factors,5).ne.2) then
            ierr = 1
            err_msg = 'Trigonometric factors are not set up'
            chckerr(err_msg)
        end if

        ! initialize
        varr = 0.0_dp

        ! sum over modes
        do id = 1,mnmax_v
            ! setup derivative factor for varf_c
            deriv_fac = mn_v(id,1)**deriv_loc(1)*(-mn_v(id,2))**deriv_loc(2)*&
                &(-1)**((sum(deriv_loc)+1)/2)

            ! add terms ~ varf_c
            if (sym_loc(1)) then
                if (mod(sum(deriv_loc),2).eq.0) then                            ! even number of derivatives
                    do kd = 1,dims(3)
                        varr(:,:,kd) = varr(:,:,kd) + &
                            &varf_c(id,kd) * deriv_fac * &
                            &trigon_factors(id,:,:,kd,1)
                    end do
                else                                                            ! odd number of derivatives
                    do kd = 1,dims(3)
                        varr(:,:,kd) = varr(:,:,kd) + &
                            &varf_c(id,kd) * deriv_fac * &
                            &trigon_factors(id,:,:,kd,2)
                    end do
                end if
            end if

            ! setup derivative factor for varf_s
            deriv_fac = mn_v(id,1)**deriv_loc(1)*(-mn_v(id,2))**deriv_loc(2)*&
                &(-1)**(sum(deriv_loc)/2)

            ! add terms ~ varf_s
            if (sym_loc(2)) then
                if (mod(sum(deriv_loc),2).eq.0) then                            ! even number of derivatives
                    do kd = 1,dims(3)
                        varr(:,:,kd) = varr(:,:,kd) + &
                            &varf_s(id,kd) * deriv_fac * &
                            &trigon_factors(id,:,:,kd,2)
                    end do
                else                                                            ! odd number of derivatives
                    do kd = 1,dims(3)
                        varr(:,:,kd) = varr(:,:,kd) + &
                            &varf_s(id,kd) * deriv_fac * &
                            &trigon_factors(id,:,:,kd,1)
                    end do
                end if
            end if
        end do
    end function fourier2real_1
    !> \private version with angles
    integer function fourier2real_2(varf_c,varf_s,theta,zeta,varr,sym,deriv) &
```

```fortran
        &result(ierr)

        character(*), parameter :: rout_name = 'fourier2real_2'

        ! input / output
        real(dp), intent(in) :: varf_c(:,:)                                     !< \f$\cos\f$ factor of variable in Fourier space
        real(dp), intent(in) :: varf_s(:,:)                                     !< \f$\sin\f$ factor of variable in Fourier space
        real(dp), intent(in) :: theta(:,:,:)                                    !< \f$\theta_\text{E}\f$
        real(dp), intent(in) :: zeta(:,:,:)                                     !< \f$\zeta_\text{E}\f$
        real(dp), intent(inout) :: varr(:,:,:)                                  !< variable in real space
        logical, intent(in), optional :: sym(2)                                 !< whether to use \c varf_c (1) and / or \c varf_s (2)
        integer, intent(in), optional :: deriv(2)                               !< optional derivatives in angular coordinates


        ! local variables
        character(len=max_str_ln) :: err_msg                                    ! error message
        integer :: dims(3)                                                      ! dimensions of varr
        integer :: id, kd                                                       ! counters
        integer :: deriv_loc(2)                                                 ! local derivative
        logical :: sym_loc(2)                                                   ! local sym
        real(dp) :: deriv_fac                                                   ! factur due to derivatives
        real(dp), allocatable :: ang(:,:,:)                                     ! angle of trigonometric functions

        ! initialize ierr
        ierr = 0

        ! set local deriv and sym
        deriv_loc = [0,0]
        sym_loc = [.true.,.true.]
        if (present(deriv)) deriv_loc = deriv
        if (present(sym)) sym_loc = sym

        ! test
        if (.not.sym_loc(1) .and. .not.sym_loc(2)) then
            ierr = 1
            err_msg = 'Need at least the cosine or the sine factor'
            chckerr(err_msg)
        end if

        ! set dimensions
        dims = shape(varr)

        ! initialize angle
        allocate(ang(dims(1),dims(2),dims(3)))

        ! initialize output
        varr = 0.0_dp

        ! sum over modes
        do id = 1,mnmax_v
            ! set angle
            ang = mn_v(id,1)*theta - mn_v(id,2)*zeta

            ! setup derivative factor for varf_c
            deriv_fac = mn_v(id,1)**deriv_loc(1)*(-mn_v(id,2))**deriv_loc(2)*&
                &(-1)**((sum(deriv_loc)+1)/2)

            ! add terms ~ varf_c
            if (sym_loc(1)) then
                if (mod(sum(deriv_loc),2).eq.0) then                            ! even number of derivatives
                    do kd = 1,dims(3)
                        varr(:,:,kd) = varr(:,:,kd) + &
                            &varf_c(id,kd) * deriv_fac * cos(ang(:,:,kd))
                    end do
                else                                                            ! odd number of derivatives
                    do kd = 1,dims(3)
                        varr(:,:,kd) = varr(:,:,kd) + &
                            &varf_c(id,kd) * deriv_fac * sin(ang(:,:,kd))
                    end do
                end if
            end if

            ! setup derivative factor for varf_s
            deriv_fac = mn_v(id,1)**deriv_loc(1)*(-mn_v(id,2))**deriv_loc(2)*&
                &(-1)**(sum(deriv_loc)/2)

            ! add terms ~ varf_s
            if (sym_loc(2)) then
                if (mod(sum(deriv_loc),2).eq.0) then                            ! even number of derivatives
                    do kd = 1,dims(3)
                        varr(:,:,kd) = varr(:,:,kd) + &
                            &varf_s(id,kd) * deriv_fac * sin(ang(:,:,kd))
                    end do
```

```
                else                                                    ! odd number of derivatives
                    do kd = 1,dims(3)
                        varr(:,:,kd) = varr(:,:,kd) + &
                            &varf_s(id,kd) * deriv_fac * cos(ang(:,:,kd))
                    end do
                end if
            end if
        end do
    end function fourier2real_2

    !> Calculate the trigonometric cosine and sine factors.
    !!
    !! This is done on a grid <tt>(1:mnmax_V)</tt> at given 3D arrays
    !! for the (VMEC) E(quilibrium) angles \f$\theta_\text{E}\f$ and
    !! \f$\zeta_\text{E}\f$.
    !!
    !! The dimensions of the output array are
    !!
    !! <tt>(1:mnmax_V,1:n_ang_1,1:n_ang_2,1:n_r,1:2)</tt>
    !!
    !! where \c mnmax_V is the number of modes in VMEC \c n_r is the total
    !! number of normal points.
    !!
    !! \see See grid_vars.grid_type for a discussion on \c ang_1 and \c ang_2.
    !!
    !! \return ierr
    integer function calc_trigon_factors(theta,zeta,trigon_factors) &
        &result(ierr)

        character(*), parameter :: rout_name = 'calc_trigon_factors'

        ! input / output
        real(dp), intent(in) :: theta(:,:,:)                            !< poloidal angles in equilibrium coords.
        real(dp), intent(in) :: zeta(:,:,:)                             !< toroidal angles in equilibrium coords.
        real(dp), intent(inout), allocatable :: trigon_factors(:,:,:,:,:)       !< trigonometric factor cosine and sine at these angles

        ! local variables
        character(len=max_str_ln) :: err_msg                            ! error message
        integer :: n_ang_1, n_ang_2, n_r                                ! sizes of 3D real output array
        real(dp), allocatable :: cos_theta(:,:,:,:)                     ! cos(m theta) for all m
        real(dp), allocatable :: sin_theta(:,:,:,:)                     ! sin(m theta) for all m
        real(dp), allocatable :: cos_zeta(:,:,:,:)                      ! cos(n theta) for all n
        real(dp), allocatable :: sin_zeta(:,:,:,:)                      ! sin(n theta) for all n
        integer :: id, m, n                                             ! counters

        ! initialize ierr
        ierr = 0

        ! set n_ang_1 and n_r
        n_ang_1 = size(theta,1)
        n_ang_2 = size(theta,2)
        n_r = size(theta,3)

#if ldebug
        if (debug_calc_trigon_factors) then
            call writo('Calculate trigonometric factors for grid of size ['//&
                &trim(i2str(n_ang_1))//','//trim(i2str(n_ang_2))//','//&
                &trim(i2str(n_r))//']')
        end if
#endif

        ! tests
        if (size(zeta,1).ne.n_ang_1 .or. size(zeta,2).ne.n_ang_2 .or. &
            &size(zeta,3).ne.n_r) then
            ierr = 1
            err_msg = 'theta and zeta need to have the same size'
            chckerr(err_msg)
        end if

        ! setup cos_theta, sin_theta, cos_zeta and sin_zeta
        allocate(cos_theta(0:mpol_v-1,n_ang_1,n_ang_2,n_r))
        allocate(sin_theta(0:mpol_v-1,n_ang_1,n_ang_2,n_r))
        allocate(cos_zeta(-ntor_v:ntor_v,n_ang_1,n_ang_2,n_r))
        allocate(sin_zeta(-ntor_v:ntor_v,n_ang_1,n_ang_2,n_r))

        do m = 0,mpol_v-1
#if ldebug
            if (debug_calc_trigon_factors) then
                call writo('Calculating for m = '//trim(i2str(m))//&
                    &' of [0..'//trim(i2str(mpol_v-1))//']')
            end if
#endif
            cos_theta(m,:,:,:) = cos(m*theta)
            sin_theta(m,:,:,:) = sin(m*theta)
        end do
        do n = -ntor_v,ntor_v
```

```
#if ldebug
            if (debug_calc_trigon_factors) then
                call writo('Calculating for n = '//trim(i2str(n))//&
                    &' of ['//trim(i2str(-ntor_v))//'..'//&
                    &trim(i2str(ntor_v))//']')
            end if
#endif
            cos_zeta(n,:,:,:) = cos(n*nfp_v*zeta)
            sin_zeta(n,:,:,:) = sin(n*nfp_v*zeta)
        end do

        ! initialize trigon_factors
        allocate(trigon_factors(mnmax_v,n_ang_1,n_ang_2,n_r,2),stat=ierr)
        chckerr('Failed to allocate trigonometric factors')
        trigon_factors = 0.0_dp

        ! calculate cos(m theta - n zeta) =
        !   cos(m theta) cos(n zeta) + sin(m theta) sin(n zeta)
        ! and sin(m theta - n zeta) =
        !   sin(m theta) cos(n zeta) - cos(m theta) sin(n zeta)
        ! Note: need to scale the indices mn_V(:,2) by nfp_V.
        do id = 1,mnmax_v
#if ldebug
            if (debug_calc_trigon_factors) then
                call writo('Calculating for id = '//trim(i2str(id))//&
                        &' of [1..'//trim(i2str(mnmax_v))//']')
            end if
#endif
            trigon_factors(id,:,:,:,1) = &
                &cos_theta(mn_v(id,1),:,:,:)*&
                &cos_zeta(mn_v(id,2)/nfp_v,:,:,:) + &
                &sin_theta(mn_v(id,1),:,:,:)*&
                &sin_zeta(mn_v(id,2)/nfp_v,:,:,:)
            trigon_factors(id,:,:,:,2) = &
                &sin_theta(mn_v(id,1),:,:,:)*&
                &cos_zeta(mn_v(id,2)/nfp_v,:,:,:) - &
                &cos_theta(mn_v(id,1),:,:,:)*&
                &sin_zeta(mn_v(id,2)/nfp_v,:,:,:)
        end do

        ! deallocate variables
        deallocate(cos_theta,sin_theta)
        deallocate(cos_zeta,sin_zeta)
    end function calc_trigon_factors
end module vmec_utilities
```

D.96 /Users/toonweyens/Code/PB3D/Modules/VMEC_vars.f90 File Reference

Modules

- module vmec_vars

  Variables that concern the output of VMEC.

Functions/Subroutines

- subroutine, public vmec_vars::dealloc_vmec ()

  Deallocates VMEC quantities that are not used anymore.

Variables

- integer, dimension(:,:), allocatable, public vmec_vars::mn_v

  m and n of modes
- real(dp), public vmec_vars::b_0_v

  the magnitude of B at the magnetic axis, $\theta = \zeta = 0$

- real(dp), dimension(:,:), allocatable, public vmec_vars::flux_t_v

  toroidal flux
- real(dp), dimension(:,:), allocatable, public vmec_vars::flux_p_v

  poloidal flux
- real(dp), dimension(:,:), allocatable, public vmec_vars::pres_v

  pressure
- real(dp), dimension(:,:), allocatable, public vmec_vars::rot_t_v

  rotational transform
- real(dp), dimension(:,:), allocatable, public vmec_vars::q_saf_v

  safety factor
- real(dp), dimension(:,:,:), allocatable, public vmec_vars::r_v_c

  Coeff. of R in sine series (FM) and norm. deriv.
- real(dp), dimension(:,:,:), allocatable, public vmec_vars::r_v_s

  Coeff. of R in cosine series (FM) and norm. deriv.
- real(dp), dimension(:,:,:), allocatable, public vmec_vars::z_v_c

  Coeff. of Z in sine series (FM) and norm. deriv.
- real(dp), dimension(:,:,:), allocatable, public vmec_vars::z_v_s

  Coeff. of Z in cosine series (FM) and norm. deriv.
- real(dp), dimension(:,:,:), allocatable, public vmec_vars::l_v_c

  Coeff. of λ in sine series (HM) and norm. deriv.
- real(dp), dimension(:,:,:), allocatable, public vmec_vars::l_v_s

  Coeff. of λ in cosine series (HM) and norm. deriv.
- real(dp), dimension(:,:,:), allocatable, public vmec_vars::jac_v_c

  Coeff. of $\mathcal{J}$ in sine series (HM and FM) and norm. deriv.
- real(dp), dimension(:,:,:), allocatable, public vmec_vars::jac_v_s

  Coeff. of $\mathcal{J}$ in cosine series (HM and FM) and norm. deriv.
- real(dp), dimension(:,:,:), allocatable, public vmec_vars::b_v_sub_c

  Coeff. of B_i in sine series (r,theta,phi) (FM).
- real(dp), dimension(:,:,:), allocatable, public vmec_vars::b_v_sub_s

  Coeff. of B_i in cosine series (r,theta,phi) (FM).
- real(dp), dimension(:,:), allocatable, public vmec_vars::b_v_c

  Coeff. of magnitude of B in sine series (HM and FM).
- real(dp), dimension(:,:), allocatable, public vmec_vars::b_v_s

  Coeff. of magnitude of B in cosine series (HM and FM).
- real(dp), dimension(:,:), allocatable, public vmec_vars::j_v_sup_int

  Integrated poloidal and toroidal current (FM).

D.97 VMEC_vars.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Variables that concern the output of VMEC.
!-----------------------------------------------------------------------------!
module vmec_vars
#include <PB3D_macros.h>
    use str_utilities
    use output_ops
    use messages
    use num_vars, only: &
        &dp, max_str_ln, pi
    use read_wout_mod, only: &                                          ! from LIBSTELL
        &is_asym_v => lasym, vmec_version => version_, is_freeb_v => lfreeb, &  ! stellerator symmetry, version number, free boundary or not
        &mpol_v => mpol, ntor_v => ntor, &                              ! mpol, ntor
```

```
        &mnmax_v => mnmax, nfp_v => nfp, &                                  ! mnmax, nfp
        &aspr_v => aspect, &                                                ! aspect ratio
        &beta_v => betaxis, &                                               ! beta on axis
        &gam_v => gamma, &                                                  ! gamma in adiabatic law (not important here, incompressibility)
        &rmax_surf, rmin_surf, zmax_surf                                    ! max and min values of R, Z

    implicit none
    private
    public dealloc_vmec, &
        &r_v_c, r_v_s, z_v_c, z_v_s, l_v_c, l_v_s, jac_v_c, jac_v_s, mnmax_v, &
        &mpol_v, ntor_v, mn_v, pres_v, rot_t_v, q_saf_v, is_asym_v, flux_t_v, &
        &flux_p_v, vmec_version, gam_v, is_freeb_v, nfp_v, b_0_v, rmin_surf, &
        &rmax_surf, aspr_v, beta_v, b_v_sub_s, b_v_sub_c
#if ldebug
    public b_v_c, b_v_s, j_v_sup_int
#endif

    ! global variables
    integer, allocatable :: mn_v(:,:)                                       !< m and n of modes
    real(dp) :: b_0_v                                                       !< the magnitude of B at the magnetic axis, \f$\theta = \zeta = 0\f$
    real(dp), allocatable :: flux_t_v(:,:)                                  !< toroidal flux
    real(dp), allocatable :: flux_p_v(:,:)                                  !< poloidal flux
    real(dp), allocatable :: pres_v(:,:)                                    !< pressure
    real(dp), allocatable :: rot_t_v(:,:)                                   !< rotational transform
    real(dp), allocatable :: q_saf_v(:,:)                                   !< safety factor
    real(dp), allocatable :: r_v_c(:,:,:)                                   !< Coeff. of \f$R\f$ in sine series (FM) and norm. deriv.
    real(dp), allocatable :: r_v_s(:,:,:)                                   !< Coeff. of \f$R\f$ in cosine series (FM) and norm. deriv.
    real(dp), allocatable :: z_v_c(:,:,:)                                   !< Coeff. of \f$Z\f$ in sine series (FM) and norm. deriv.
    real(dp), allocatable :: z_v_s(:,:,:)                                   !< Coeff. of \f$Z\f$ in cosine series (FM) and norm. deriv.
    real(dp), allocatable :: l_v_c(:,:,:)                                   !< Coeff. of \f$\lambda\f$ in sine series (HM) and norm. deriv.
    real(dp), allocatable :: l_v_s(:,:,:)                                   !< Coeff. of \f$\lambda\f$ in cosine series (HM) and norm. deriv.
    real(dp), allocatable :: jac_v_c(:,:,:)                                 !< Coeff. of \f$\mathcal{J}\f$ in sine series (HM and FM) and norm. deriv.
    real(dp), allocatable :: jac_v_s(:,:,:)                                 !< Coeff. of \f$\mathcal{J}\f$ in cosine series (HM and FM) and norm. deriv.
    real(dp), allocatable :: b_v_sub_c(:,:,:)                               !< Coeff. of B_i in sine series (r,theta,phi) (FM) \ldebug
    real(dp), allocatable :: b_v_sub_s(:,:,:)                               !< Coeff. of B_i in cosine series (r,theta,phi) (FM) \ldebug
#if ldebug
    real(dp), allocatable :: b_v_c(:,:)                                     !< Coeff. of magnitude of B in sine series (HM and FM) \ldebug
    real(dp), allocatable :: b_v_s(:,:)                                     !< Coeff. of magnitude of B in cosine series (HM and FM) \ldebug
    real(dp), allocatable :: j_v_sup_int(:,:)                               !< Integrated poloidal and toroidal current (FM) \ldebug
#endif
contains
    !> Deallocates VMEC quantities that are not used anymore.
    subroutine dealloc_vmec()
#if ldebug
        use num_vars, only: rank, print_mem_usage

        ! local variables
        integer :: mem_diff                                                 ! difference in memory

        ! memory usage before deallocation
        if (print_mem_usage) mem_diff = get_mem_usage()
#endif

        ! deallocate
        deallocate(rot_t_v)
        deallocate(q_saf_v)
        deallocate(pres_v)
        deallocate(flux_t_v)
        deallocate(flux_p_v)
        deallocate(mn_v)
        deallocate(r_v_c)
        deallocate(r_v_s)
        deallocate(z_v_c)
        deallocate(z_v_s)
        deallocate(l_v_c)
        deallocate(l_v_s)
        deallocate(jac_v_c)
        deallocate(jac_v_s)
        deallocate(b_v_sub_c)
        deallocate(b_v_sub_s)
#if ldebug
        deallocate(b_v_c)
```

```
        deallocate(b_v_s)
        deallocate(j_v_sup_int)

        ! memory usage difference after deallocation
        if (print_mem_usage) then
            mem_diff = mem_diff - get_mem_usage()
            call writo('Rank '//trim(i2str(rank))//' liberated '//&
                &trim(i2str(mem_diff))//'kB deallocating VMEC')
        end if
#endif
    end subroutine dealloc_vmec
end module vmec_vars
```

D.98 /Users/toonweyens/Code/PB3D/Modules/X_ops.f90 File Reference

Interfaces and Types

- interface x_ops::calc_x

  Calculates either vectorial or tensorial perturbation variables.
- interface x_ops::redistribute_output_x

  Redistribute the perturbation variables.
- interface x_ops::print_output_x

  Print either vectorial or tensorial perturbation quantities of a certain order to an output file.

Modules

- module x_ops

  Operations considering perturbation quantities.

Functions/Subroutines

- integer function, public x_ops::init_modes (grid_eq, eq)

  Initializes some variables concerning the mode numbers.
- integer function, public x_ops::setup_modes (mds, grid_eq, grid, plot_name)

  Sets up some variables concerning the mode numbers.
- integer function, public x_ops::check_x_modes (grid_eq, eq)

  Checks whether the high-n approximation is valid:
- integer function, public x_ops::calc_res_surf (mds, grid_eq, eq, res_surf, info, jq)

  Calculates resonating flux surfaces for the perturbation modes.
- integer function, public x_ops::resonance_plot (mds, grid_eq, eq)

  plot q-profile or ι-profile in flux coordinates with $nq - m = 0$ or $n - \iota m = 0$ indicated if requested.
- integer function test_du ()
- integer function, public x_ops::calc_magn_ints (grid_eq, grid_x, eq, x, x_int, prev_style, lim_sec_x)

  Calculate the magnetic integrals from $\widetilde{PV}^i_{k,m}$ and $\widetilde{KV}^i_{k,m}$ in an equidistant grid where the step size can vary depending on the normal coordinate.
- integer function, public x_ops::divide_x_jobs (arr_size)

  Divides the perturbation jobs.
- integer function, public x_ops::print_debug_x_1 (mds, grid_x, x_1)

  Prints debug information for X_1 driver.
- integer function, public x_ops::print_debug_x_2 (mds, grid_x, x_2_int)

  Prints debug information for X_2 driver.

Variables

- logical, public x_ops::debug_check_x_modes_2 = .false.

  plot debug information for check_x_modes_2()

D.98.1 Function/Subroutine Documentation

D.98.1.1 test_du()

```
integer function calc_u::test_du
```

Definition at line 2514 of file X_ops.f90.

Here is the call graph for this function:

Here is the caller graph for this function:

D.99 X_ops.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Operations considering perturbation quantities.
!-----------------------------------------------------------------------------!
module x_ops
#include <PB3D_macros.h>
#include <wrappers.h>
    use str_utilities
    use output_ops
```

```
    use messages
    use num_vars, only: dp, iu, max_str_ln, max_name_ln, pi
    use grid_vars, onlY: grid_type
    use eq_vars, only: eq_1_type, eq_2_type
    use x_vars, only: x_1_type, x_2_type, modes_type

    implicit none
    private
    public calc_x, calc_magn_ints, print_output_x, resonance_plot, &
        &calc_res_surf, check_x_modes, init_modes, setup_modes, divide_x_jobs, &
        &redistribute_output_x
#if ldebug
    public debug_check_x_modes_2, print_debug_x_1, print_debug_x_2
#endif

    ! global variables
#if ldebug
    logical :: debug_check_x_modes_2 = .false.                                  !< plot debug information for check_x_modes_2() \ldebug
    logical :: debug_setup_modes = .false.                                      !< plot debug information for setup_modes() \ldebug
#endif

    ! interfaces

    !> \public Calculates either vectorial or tensorial perturbation variables.
    !!
    !! Optionally, the secondary mode number can  be specified (\c m if poloidal
    !! flux is used as normal coordinate and  c n if toroidal flux). By default,
    !! they are taken from the global \c X_vars variables.
    !!
    !! \return ierr
    interface calc_x
        !> \public
        module procedure calc_x_1
        !> \public
        module procedure calc_x_2
    end interface

    !> \public Redistribute the perturbation variables.
    !!
    !! \see redistribute_output_grid()
    !!
    !! \return ierr
    interface redistribute_output_x
        !> \public
        module procedure redistribute_output_x_1
        !> \public
        module procedure redistribute_output_x_2
    end interface

    !> \public Print either vectorial or  tensorial perturbation quantities of a
    !! certain order to an output file.
    !!
    !!  - vectorial:
    !!      - U
    !!      - DU
    !!  - tensorial:
    !!      - PV_int
    !!      - KV_int
    !!
    !! (The  non-integrated variables are not  saved because they are  heavy and
    !! not requested.)
    !!
    !! If \c  rich_lvl is provided, \c  "_R_[rich_lvl]" is appended to  the data
    !! name if it is \c >0.
    !!
    !! Optionally,  it  can be  specified that  this is  a divided
    !! parallel grid, corresponding  to the variable \c  eq_jobs_lims with index
    !! \c eq_job_nr. In  this case, the total  grid size is adjusted  to the one
    !! specified by \c eq_jobs_lims and the grid is written as a subset.
    !! \note
    !!  -# The equilibrium quantities are outputted in Flux coordinates.
    !!  -# The  tensorial perturbation type  can also be used  for field-aligned
    !!  variables, in which case the first  index is assumed to have dimension 1
    !!  only. This can be triggered using \c is_field_averaged.
    !!
    !! \return ierr
    interface print_output_x
        !> \public
        module procedure print_output_x_1
        !> \public
        module procedure print_output_x_2
    end interface

contains
    !> \private vectorial version
```

```
    integer function calc_x_1(mds,grid_eq,grid_X,eq_1,eq_2,X,lim_sec_X) &
        &result(ierr)

        character(*), parameter :: rout_name = 'calc_X_1'

        ! input / output
        type(modes_type), intent(in) :: mds                                  !< general modes variables
        type(grid_type), intent(in) :: grid_eq                               !< equilibrium grid variables
        type(grid_type), intent(in) :: grid_x                                !< perturbation grid variables
        type(eq_1_type), intent(in) :: eq_1                                  !< flux equilibrium
        type(eq_2_type), intent(in) :: eq_2                                  !< metric equilibrium
        type(x_1_type), intent(inout) :: x                                   !< vectorial perturbation variables
        integer, intent(in), optional :: lim_sec_x(2)                        !< limits of \c m_X (pol. flux) or \ n_X (tor. flux)

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Calculate vectorial perturbation variables')
        call lvl_ud(1)

        ! create perturbation with modes of current X job
        call x%init(mds,grid_x,lim_sec_x)

        ! calculate U and DU
        call writo('Calculating U and DU...')
        call lvl_ud(1)
        ierr = calc_u(grid_eq,grid_x,eq_1,eq_2,x)
        chckerr('')
        call lvl_ud(-1)

        ! user output
        call lvl_ud(-1)
    end function calc_x_1
    !> \private tensorial version
    integer function calc_x_2(mds,grid_eq,grid_X,eq_1,eq_2,X_a,X_b,X,&
        &lim_sec_X) result(ierr)

        character(*), parameter :: rout_name = 'calc_X_2'

        ! input / output
        type(modes_type), intent(in) :: mds                                  !< general modes variables
        type(grid_type), intent(in) :: grid_eq                               !< equilibrium grid variables
        type(grid_type), intent(in) :: grid_x                                !< perturbation grid variables
        type(eq_1_type), intent(in) :: eq_1                                  !< flux equilibrium
        type(eq_2_type), intent(in) :: eq_2                                  !< metric equilibrium
        type(x_1_type), intent(inout) :: x_a                                 !< vectorial perturbation variables (dimension 1)
        type(x_1_type), intent(inout) :: x_b                                 !< vectorial perturbation variables (dimension 2)
        type(x_2_type), intent(inout) :: x                                   !< tensorial perturbation variables
        integer, intent(in), optional :: lim_sec_x(2,2)                      !< limits of \c m_X (pol flux) or \c n_X (tor flux) for both dimensions

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Calculate tensorial perturbation variables')
        call lvl_ud(1)

        ! create perturbation with modes of current X job
        call x%init(mds,grid_x,lim_sec_x)

        ! Calculate  PV_i  for  all (k,m)  pairs  and n_r  (equilibrium)
        ! values of the normal coordinate
        call writo('Calculating PV...')
        call lvl_ud(1)
        ierr = calc_pv(grid_eq,grid_x,eq_1,eq_2,x_a,x_b,x,lim_sec_x)
        chckerr('')
        call lvl_ud(-1)

        !  Calculate KV_i  for  all (k,m)  pairs  and n_r  (equilibrium)
        ! values of the normal coordinate
        call writo('Calculating KV...')
        call lvl_ud(1)
        ierr = calc_kv(grid_eq,grid_x,eq_1,eq_2,x_a,x_b,x,lim_sec_x)
        chckerr('')
        call lvl_ud(-1)

        ! user output
        call lvl_ud(-1)
    end function calc_x_2

    !> \private flux version
    integer function redistribute_output_x_1(mds,grid,grid_out,X,X_out) &
```

```
        &result(ierr)
        use mpi_utilities, only: redistribute_var

        character(*), parameter :: rout_name = 'redistribute_output_X_1'

        ! input / output
        type(modes_type), intent(in) :: mds                                     !< general modes variables
        type(grid_type), intent(in) :: grid                                     !< perturbation grid variables
        type(grid_type), intent(in) :: grid_out                                 !< redistributed perturbation grid variables
        type(x_1_type), intent(in) :: x                                         !< vectorial perturbation variables
        type(x_1_type), intent(inout) :: x_out                                  !< vectorial perturbation variables in redistributed grid

        ! local variables
        integer :: ld                                                           ! counter
        integer :: lims(2), lims_dis(2)                                         ! limits and distributed limits, taking into account the angular extent
        integer :: siz(3), siz_dis(3)                                           ! size for geometric part of variable
        real(dp), allocatable :: temp_var(:)                                    ! temporary variable

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Redistribute vectorial perturbation variables')
        call lvl_ud(1)

        ! test
        if (grid%n(1).ne.grid_out%n(1) .or. grid%n(2).ne.grid_out%n(2)) then
            ierr = 1
            chckerr('grid and grid_out are not compatible')
        end if

        ! create redistributed vectorial perturbation variables
        call x_out%init(mds,grid_out,lim_sec_x=x%lim_sec_X)

        ! set up limits taking into account angular extent and temporary var
        lims(1) = product(grid%n(1:2))*(grid%i_min-1)+1
        lims(2) = product(grid%n(1:2))*grid%i_max
        lims_dis(1) = product(grid%n(1:2))*(grid_out%i_min-1)+1
        lims_dis(2) = product(grid%n(1:2))*grid_out%i_max
        siz = [grid%n(1:2),grid%loc_n_r]
        siz_dis = [grid%n(1:2),grid_out%loc_n_r]
        allocate(temp_var(product(siz_dis)))

        ! for all mode numbers
        do ld = 1,size(x%U_0,4)
            ! U_0
            ierr = redistribute_var(reshape(rp(x%U_0(:,:,:,ld)),&
                &[product(siz)]),temp_var,lims,lims_dis)
            chckerr('')
            x_out%U_0(:,:,:,ld) = &
                &reshape(temp_var(1:product(siz_dis)),siz_dis)
            ierr = redistribute_var(reshape(ip(x%U_0(:,:,:,ld)),&
                &[product(siz)]),temp_var,lims,lims_dis)
            chckerr('')
            x_out%U_0(:,:,:,ld) = x_out%U_0(:,:,:,ld) + iu*&
                &reshape(temp_var(1:product(siz_dis)),siz_dis)

            ! U_1
            ierr = redistribute_var(reshape(rp(x%U_1(:,:,:,ld)),&
                &[product(siz)]),temp_var,lims,lims_dis)
            chckerr('')
            x_out%U_1(:,:,:,ld) = &
                &reshape(temp_var(1:product(siz_dis)),siz_dis)
            ierr = redistribute_var(reshape(ip(x%U_1(:,:,:,ld)),&
                &[product(siz)]),temp_var,lims,lims_dis)
            chckerr('')
            x_out%U_1(:,:,:,ld) = x_out%U_1(:,:,:,ld) + iu*&
                &reshape(temp_var(1:product(siz_dis)),siz_dis)

            ! DU_0
            ierr = redistribute_var(reshape(rp(x%DU_0(:,:,:,ld)),&
                &[product(siz)]),temp_var,lims,lims_dis)
            chckerr('')
            x_out%DU_0(:,:,:,ld) = &
                &reshape(temp_var(1:product(siz_dis)),siz_dis)
            ierr = redistribute_var(reshape(ip(x%DU_0(:,:,:,ld)),&
                &[product(siz)]),temp_var,lims,lims_dis)
            chckerr('')
            x_out%DU_0(:,:,:,ld) = x_out%DU_0(:,:,:,ld) + iu*&
                &reshape(temp_var(1:product(siz_dis)),siz_dis)

            ! DU_1
            ierr = redistribute_var(reshape(rp(x%DU_1(:,:,:,ld)),&
                &[product(siz)]),temp_var,lims,lims_dis)
            chckerr('')
```

```
            x_out%DU_1(:,:,:,ld) = &
                &reshape(temp_var(1:product(siz_dis)),siz_dis)
            ierr = redistribute_var(reshape(ip(x%DU_1(:,:,:,ld)),&
                &[product(siz)]),temp_var,lims,lims_dis)
            chckerr('')
            x_out%DU_1(:,:,:,ld) = x_out%DU_1(:,:,:,ld) + iu*&
                &reshape(temp_var(1:product(siz_dis)),siz_dis)
        end do

        ! user output
        call lvl_ud(-1)
    end function redistribute_output_x_1
    !> \private metric version
    integer function redistribute_output_x_2(mds,grid,grid_out,X,X_out) &
        &result(ierr)
        use mpi_utilities, only: redistribute_var

        character(*), parameter :: rout_name = 'redistribute_output_X_2'

        ! input / output
        type(modes_type), intent(in) :: mds                                     !< general modes variables
        type(grid_type), intent(in) :: grid                                     !< perturbation grid variables
        type(grid_type), intent(in) :: grid_out                                 !< redistributed perturbation grid variables
        type(x_2_type), intent(in) :: x                                         !< tensorial perturbation variables
        type(x_2_type), intent(inout) :: x_out                                  !< tensorial perturbation variables in redistributed grid

        ! local variables
        integer :: ld                                                           ! counters
        integer :: n_loc(2)                                                     ! local n
        integer :: lims(2), lims_dis(2)                                         ! limits and distributed limits, taking into account the angular extent
        integer :: siz(3), siz_dis(3)                                           ! size for geometric part of variable
        logical :: is_field_averaged                                            ! whether field-averaged, so only one dimension for first index
        real(dp), allocatable :: temp_var(:)                                    ! temporary variable

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Redistribute tensorial perturbation variables')
        call lvl_ud(1)

        ! test
        is_field_averaged = size(x%PV_0,1).eq.1
        if ((grid%n(1).ne.grid_out%n(1) .or. grid%n(2).ne.grid_out%n(2)) .and. &
            &.not.is_field_averaged) then                                       ! for field-averaged grids, don't do tests
            ierr = 1
            chckerr('grid and grid_out are not compatible')
        end if

        ! create redistributed tensorial perturbation variables
        call x_out%init(mds,grid_out,lim_sec_x=x%lim_sec_X,&
            &is_field_averaged=is_field_averaged)

        ! set up limits taking into account angular extent and temporary var
        n_loc = grid%n(1:2)
        if (is_field_averaged) n_loc(1) = 1
        lims(1) = product(n_loc)*(grid%i_min-1)+1
        lims(2) = product(n_loc)*grid%i_max
        lims_dis(1) = product(n_loc)*(grid_out%i_min-1)+1
        lims_dis(2) = product(n_loc)*grid_out%i_max
        siz = [n_loc,grid%loc_n_r]
        siz_dis = [n_loc,grid_out%loc_n_r]
        allocate(temp_var(product(siz_dis)))

        ! for all mode number combinations, symmetric
        do ld = 1,size(x%PV_0,4)
            ! PV_0
            ierr = redistribute_var(reshape(rp(x%PV_0(1:n_loc(1),:,:,ld)),&
                &[product(siz)]),temp_var,lims,lims_dis)
            chckerr('')
            x_out%PV_0(1:n_loc(1),:,:,ld) = &
                &reshape(temp_var(1:product(siz_dis)),siz_dis)
            ierr = redistribute_var(reshape(ip(x%PV_0(1:n_loc(1),:,:,ld)),&
                &[product(siz)]),temp_var,lims,lims_dis)
            chckerr('')
            x_out%PV_0(1:n_loc(1),:,:,ld) = x_out%PV_0(1:n_loc(1),:,:,ld) + iu*&
                &reshape(temp_var(1:product(siz_dis)),siz_dis)

            ! PV_2
            ierr = redistribute_var(reshape(rp(x%PV_2(1:n_loc(1),:,:,ld)),&
                &[product(siz)]),temp_var,lims,lims_dis)
            chckerr('')
            x_out%PV_2(1:n_loc(1),:,:,ld) = &
                &reshape(temp_var(1:product(siz_dis)),siz_dis)
```

```
            ierr = redistribute_var(reshape(ip(x%PV_2(1:n_loc(1),:,:,ld)),&
                &[product(siz)]),temp_var,lims,lims_dis)
            chckerr('')
            x_out%PV_2(1:n_loc(1),:,:,ld) = x_out%PV_2(1:n_loc(1),:,:,ld) + iu*&
                &reshape(temp_var(1:product(siz_dis)),siz_dis)

            ! KV_0
            ierr = redistribute_var(reshape(rp(x%KV_0(1:n_loc(1),:,:,ld)),&
                &[product(siz)]),temp_var,lims,lims_dis)
            chckerr('')
            x_out%KV_0(1:n_loc(1),:,:,ld) = &
                &reshape(temp_var(1:product(siz_dis)),siz_dis)
            ierr = redistribute_var(reshape(ip(x%KV_0(1:n_loc(1),:,:,ld)),&
                &[product(siz)]),temp_var,lims,lims_dis)
            chckerr('')
            x_out%KV_0(1:n_loc(1),:,:,ld) = x_out%KV_0(1:n_loc(1),:,:,ld) + iu*&
                &reshape(temp_var(1:product(siz_dis)),siz_dis)

            ! KV_2
            ierr = redistribute_var(reshape(rp(x%KV_2(1:n_loc(1),:,:,ld)),&
                &[product(siz)]),temp_var,lims,lims_dis)
            chckerr('')
            x_out%KV_2(1:n_loc(1),:,:,ld) = &
                &reshape(temp_var(1:product(siz_dis)),siz_dis)
            ierr = redistribute_var(reshape(ip(x%KV_2(1:n_loc(1),:,:,ld)),&
                &[product(siz)]),temp_var,lims,lims_dis)
            chckerr('')
            x_out%KV_2(1:n_loc(1),:,:,ld) = x_out%KV_2(1:n_loc(1),:,:,ld) + iu*&
                &reshape(temp_var(1:product(siz_dis)),siz_dis)
        end do

        ! for all mode number combinations, asymmetric
        do ld = 1,size(x%PV_1,4)
            ! PV_1
            ierr = redistribute_var(reshape(rp(x%PV_1(1:n_loc(1),:,:,ld)),&
                &[product(siz)]),temp_var,lims,lims_dis)
            chckerr('')
            x_out%PV_1(1:n_loc(1),:,:,ld) = &
                &reshape(temp_var(1:product(siz_dis)),siz_dis)
            ierr = redistribute_var(reshape(ip(x%PV_1(1:n_loc(1),:,:,ld)),&
                &[product(siz)]),temp_var,lims,lims_dis)
            chckerr('')
            x_out%PV_1(1:n_loc(1),:,:,ld) = x_out%PV_1(1:n_loc(1),:,:,ld) + iu*&
                &reshape(temp_var(1:product(siz_dis)),siz_dis)

            ! KV_1
            ierr = redistribute_var(reshape(rp(x%KV_1(1:n_loc(1),:,:,ld)),&
                &[product(siz)]),temp_var,lims,lims_dis)
            chckerr('')
            x_out%KV_1(1:n_loc(1),:,:,ld) = &
                &reshape(temp_var(1:product(siz_dis)),siz_dis)
            ierr = redistribute_var(reshape(ip(x%KV_1(1:n_loc(1),:,:,ld)),&
                &[product(siz)]),temp_var,lims,lims_dis)
            chckerr('')
            x_out%KV_1(1:n_loc(1),:,:,ld) = x_out%KV_1(1:n_loc(1),:,:,ld) + iu*&
                &reshape(temp_var(1:product(siz_dis)),siz_dis)
        end do

        ! user output
        call lvl_ud(-1)
    end function redistribute_output_x_2

    !> \private vectorial version.
    integer function print_output_x_1(grid_X,X,data_name,rich_lvl,par_div,&
        &lim_sec_X) result(ierr)

        use num_vars, only: pb3d_name, eq_jobs_lims, eq_job_nr
        use grid_utilities, only: trim_grid
        use hdf5_ops, only: print_hdf5_arrs
        use hdf5_vars, only: dealloc_var_1d, var_1d_type, &
            &max_dim_var_1d
        use x_vars, only: n_mod_x

        character(*), parameter :: rout_name = 'print_output_X_1'

        ! input / output
        type(grid_type), intent(in) :: grid_x                                   !< perturbation grid variables
        type(x_1_type), intent(in) :: x                                         !< vectorial perturbation variables
        character(len=*), intent(in) :: data_name                               !< name under which to store
        integer, intent(in), optional :: rich_lvl                               !< Richardson level to print
        logical, intent(in), optional :: par_div                                !< is a parallely divided grid
        integer, intent(in), optional :: lim_sec_x(2)                           !< limits of \c m_X (pol. flux) or \c n_X
(tor. flux)

        ! local variables
        type(grid_type) :: grid_trim                                            ! trimmed grid
        integer :: n_tot(3)                                                     ! total n
```

```
        integer :: par_id(2)                                                    ! parallel interval
        integer :: norm_id(2)                                                   ! untrimmed normal indices for trimmed grids
        type(var_1d_type), allocatable, target :: x_1d(:)                       ! 1D equivalent of X variables
        type(var_1d_type), pointer :: x_1d_loc => null()                        ! local element in X_1D
        integer :: id                                                           ! counters
        logical :: par_div_loc                                                  ! local par_div
        integer :: lim_sec_x_loc(2)                                             ! local lim_sec_X
        integer :: loc_size                                                     ! local size

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Write vectorial perturbation variables to output file')
        call lvl_ud(1)

        ! trim grid
        ierr = trim_grid(grid_x,grid_trim,norm_id)
        chckerr('')

        ! set local par_div
        par_div_loc = .false.
        if (present(par_div)) par_div_loc = par_div

        ! set total n and parallel interval
        n_tot = grid_trim%n
        par_id = [1,n_tot(1)]
        if (grid_trim%n(1).gt.0 .and. par_div_loc) then                         ! total grid includes all equilibrium jobs
            n_tot(1) = maxval(eq_jobs_lims)-minval(eq_jobs_lims)+1
            par_id = eq_jobs_lims(:,eq_job_nr)
        end if

        ! set local size and lim_sec_X
        loc_size = size(x%U_0(:,:,norm_id(1):norm_id(2),:))
        lim_sec_x_loc = [1,n_mod_x]
        if (present(lim_sec_x)) lim_sec_x_loc = lim_sec_x

        ! Set up the 1D equivalents of the perturbation variables
        allocate(x_1d(max_dim_var_1d))

        ! set up variables X_1D
        id = 1

        ! RE_U_0
        x_1d_loc => x_1d(id); id = id+1
        x_1d_loc%var_name = 'RE_U_0'
        allocate(x_1d_loc%tot_i_min(4),x_1d_loc%tot_i_max(4))
        allocate(x_1d_loc%loc_i_min(4),x_1d_loc%loc_i_max(4))
        x_1d_loc%tot_i_min = [1,1,1,1]
        x_1d_loc%tot_i_max = [n_tot,n_mod_x]
        x_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,lim_sec_x_loc(1)]
        x_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,&
            &lim_sec_x_loc(2)]
        allocate(x_1d_loc%p(loc_size))
        x_1d_loc%p = reshape(rp(x%U_0(:,:,norm_id(1):norm_id(2),:)),&
            &[loc_size])

        ! IM_U_0
        x_1d_loc => x_1d(id); id = id+1
        x_1d_loc%var_name = 'IM_U_0'
        allocate(x_1d_loc%tot_i_min(4),x_1d_loc%tot_i_max(4))
        allocate(x_1d_loc%loc_i_min(4),x_1d_loc%loc_i_max(4))
        x_1d_loc%tot_i_min = [1,1,1,1]
        x_1d_loc%tot_i_max = [n_tot,n_mod_x]
        x_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,lim_sec_x_loc(1)]
        x_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,&
            &lim_sec_x_loc(2)]
        allocate(x_1d_loc%p(loc_size))
        x_1d_loc%p = reshape(ip(x%U_0(:,:,norm_id(1):norm_id(2),:)),&
            &[loc_size])

        ! RE_U_1
        x_1d_loc => x_1d(id); id = id+1
        x_1d_loc%var_name = 'RE_U_1'
        allocate(x_1d_loc%tot_i_min(4),x_1d_loc%tot_i_max(4))
        allocate(x_1d_loc%loc_i_min(4),x_1d_loc%loc_i_max(4))
        x_1d_loc%tot_i_min = [1,1,1,1]
        x_1d_loc%tot_i_max = [n_tot,n_mod_x]
        x_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,lim_sec_x_loc(1)]
        x_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,&
            &lim_sec_x_loc(2)]
        allocate(x_1d_loc%p(loc_size))
        x_1d_loc%p = reshape(rp(x%U_1(:,:,norm_id(1):norm_id(2),:)),&
            &[loc_size])

        ! IM_U_1
        x_1d_loc => x_1d(id); id = id+1
```

```
        x_1d_loc%var_name = 'IM_U_1'
        allocate(x_1d_loc%tot_i_min(4),x_1d_loc%tot_i_max(4))
        allocate(x_1d_loc%loc_i_min(4),x_1d_loc%loc_i_max(4))
        x_1d_loc%tot_i_min = [1,1,1,1]
        x_1d_loc%tot_i_max = [n_tot,n_mod_x]
        x_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,lim_sec_x_loc(1)]
        x_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,&
            &lim_sec_x_loc(2)]
        allocate(x_1d_loc%p(loc_size))
        x_1d_loc%p = reshape(ip(x%U_1(:,:,norm_id(1):norm_id(2),:)),&
            &[loc_size])

        ! RE_DU_0
        x_1d_loc => x_1d(id); id = id+1
        x_1d_loc%var_name = 'RE_DU_0'
        allocate(x_1d_loc%tot_i_min(4),x_1d_loc%tot_i_max(4))
        allocate(x_1d_loc%loc_i_min(4),x_1d_loc%loc_i_max(4))
        x_1d_loc%tot_i_min = [1,1,1,1]
        x_1d_loc%tot_i_max = [n_tot,n_mod_x]
        x_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,lim_sec_x_loc(1)]
        x_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,&
            &lim_sec_x_loc(2)]
        allocate(x_1d_loc%p(loc_size))
        x_1d_loc%p = reshape(rp(x%DU_0(:,:,norm_id(1):norm_id(2),:)),&
            &[loc_size])

        ! IM_DU_0
        x_1d_loc => x_1d(id); id = id+1
        x_1d_loc%var_name = 'IM_DU_0'
        allocate(x_1d_loc%tot_i_min(4),x_1d_loc%tot_i_max(4))
        allocate(x_1d_loc%loc_i_min(4),x_1d_loc%loc_i_max(4))
        x_1d_loc%tot_i_min = [1,1,1,1]
        x_1d_loc%tot_i_max = [n_tot,n_mod_x]
        x_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,lim_sec_x_loc(1)]
        x_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,&
            &lim_sec_x_loc(2)]
        allocate(x_1d_loc%p(loc_size))
        x_1d_loc%p = reshape(ip(x%DU_0(:,:,norm_id(1):norm_id(2),:)),&
            &[loc_size])

        ! RE_DU_1
        x_1d_loc => x_1d(id); id = id+1
        x_1d_loc%var_name = 'RE_DU_1'
        allocate(x_1d_loc%tot_i_min(4),x_1d_loc%tot_i_max(4))
        allocate(x_1d_loc%loc_i_min(4),x_1d_loc%loc_i_max(4))
        x_1d_loc%tot_i_min = [1,1,1,1]
        x_1d_loc%tot_i_max = [n_tot,n_mod_x]
        x_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,lim_sec_x_loc(1)]
        x_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,&
            &lim_sec_x_loc(2)]
        allocate(x_1d_loc%p(loc_size))
        x_1d_loc%p = reshape(rp(x%DU_1(:,:,norm_id(1):norm_id(2),:)),&
            &[loc_size])

        ! IM_DU_1
        x_1d_loc => x_1d(id); id = id+1
        x_1d_loc%var_name = 'IM_DU_1'
        allocate(x_1d_loc%tot_i_min(4),x_1d_loc%tot_i_max(4))
        allocate(x_1d_loc%loc_i_min(4),x_1d_loc%loc_i_max(4))
        x_1d_loc%tot_i_min = [1,1,1,1]
        x_1d_loc%tot_i_max = [n_tot,n_mod_x]
        x_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,lim_sec_x_loc(1)]
        x_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,&
            &lim_sec_x_loc(2)]
        allocate(x_1d_loc%p(loc_size))
        x_1d_loc%p = reshape(ip(x%DU_1(:,:,norm_id(1):norm_id(2),:)),&
            &[loc_size])

        ! write
        ierr = print_hdf5_arrs(x_1d(1:id-1),pb3d_name,trim(data_name),&
            &rich_lvl=rich_lvl,ind_print=.not.grid_trim%divided)
        chckerr('')

        ! clean up
        call grid_trim%dealloc()
        call dealloc_var_1d(x_1d)
        nullify(x_1d_loc)

        ! user output
        call lvl_ud(-1)
    end function print_output_x_1
    !> \private tensorial version
    integer function print_output_x_2(grid_X,X,data_name,rich_lvl,par_div,&
        &lim_sec_X,is_field_averaged) result(ierr)
        use num_vars, only: pb3d_name, eq_jobs_lims, eq_job_nr
        use grid_utilities, only: trim_grid
        use hdf5_ops, only: print_hdf5_arrs
```

```
        use hdf5_vars, only: dealloc_var_1d, var_1d_type, &
            &max_dim_var_1d
        use x_utilities, only: is_necessary_x, get_sec_x_range
        use x_vars, only: set_nn_mod
        use num_utilities, only: c

        character(*), parameter :: rout_name = 'print_output_X_2'

        ! input / output
        type(grid_type), intent(in) :: grid_x                                   !< perturbation grid variables
        type(x_2_type), intent(in) :: x                                         !< tensorial perturbation variables
        character(len=*), intent(in) :: data_name                               !< name under which to store
        integer, intent(in), optional :: rich_lvl                               !< Richardson level to print
        logical, intent(in), optional :: par_div                                !< is a parallely divided grid
        integer, intent(in), optional :: lim_sec_x(2,2)                         !< limits of \c m_X (pol. flux) or \c n_X (tor. flux)
        logical, intent(in), optional :: is_field_averaged                      !< whether field-averaged, so only one dimension for first index

        ! local variables
        type(grid_type) :: grid_trim                                            ! trimmed grid
        integer :: n_tot(3)                                                     ! total n
        integer :: par_id(2)                                                    ! parallel interval
        integer :: par_id_loc(2)                                                ! local parallel interval
        integer :: norm_id(2)                                                   ! untrimmed normal indices for trimmed grids
        type(var_1d_type), allocatable, target :: x_1d(:)                       ! 1D equivalent of X variables
        type(var_1d_type), pointer :: x_1d_loc => null()                        ! local element in X_1D
        logical :: print_this(2)                                                ! whether symmetric and asymmetric variables need to be printed
        integer :: nn_mod_tot(2)                                                ! total nr. of modes for symmetric and asymmetric variables
        integer :: nn_mod_loc(2)                                                ! local nr. of modes for symmetric and asymmetric variables
        integer :: id                                                           ! counter
        logical :: par_div_loc                                                  ! local par_div
        integer :: loc_size(2)                                                  ! local size for symmetric and asymmetric variables
        integer :: m, k                                                         ! counters
        integer :: sxr_loc(2,2)                                                 ! local secondary X limits for symmetric and asymmetric variables
        integer :: sxr_tot(2,2)                                                 ! total secondary X limits for symmetric and asymmetric variables

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Write tensorial perturbation variables to output file')
        call lvl_ud(1)

        ! trim grid
        ierr = trim_grid(grid_x,grid_trim,norm_id)
        chckerr('')

        ! set local par_div
        par_div_loc = .false.
        if (present(par_div)) par_div_loc = par_div

        ! set total n, parallel interval and total n_mod_X
        n_tot = grid_trim%n
        par_id = [1,n_tot(1)]
        par_id_loc = [1,n_tot(1)]
        if (grid_trim%n(1).gt.0 .and. par_div_loc) then                         ! total grid includes all equilibrium jobs
            n_tot(1) = maxval(eq_jobs_lims)-minval(eq_jobs_lims)+1
            par_id = eq_jobs_lims(:,eq_job_nr)
            par_id_loc = par_id - par_id(1)+1
        else if (present(is_field_averaged)) then
            if (is_field_averaged) then                                         ! only first point
                par_id = [1,1]
                par_id_loc = [1,1]
                n_tot(1) = 1
            end if
        end if
        nn_mod_tot(1) = set_nn_mod(.true.)
        nn_mod_tot(2) = set_nn_mod(.false.)

        ! set dimensions, parallel limits and local and total nn_mod_X

        ! Set up the 1D equivalents of the perturbation variables
        allocate(x_1d(max_dim_var_1d))

        ! set up variables X_1D
        id = 1

        ! loop over modes of dimension 2
        do m = 1,x%n_mod(2)
            ! get contiguous range of modes of this m
```

```
            call get_sec_x_range(sxr_loc(:,1),sxr_tot(:,1),m,.true.,lim_sec_x)
            call get_sec_x_range(sxr_loc(:,2),sxr_tot(:,2),m,.false.,lim_sec_x)
            nn_mod_loc = sxr_loc(2,:)-sxr_loc(1,:)+1
            print_this = .false.
            do k = 1,2
                if (sxr_loc(1,k).le.sxr_loc(2,k)) print_this(k) = .true.        ! a bound is found
            end do

            ! set local size
            loc_size(1) = (par_id(2)-par_id(1)+1)*n_tot(2)*&
                &(norm_id(2)-norm_id(1)+1)*nn_mod_loc(1)
            loc_size(2) = (par_id(2)-par_id(1)+1)*n_tot(2)*&
                &(norm_id(2)-norm_id(1)+1)*nn_mod_loc(2)

            if (print_this(1)) then
                ! RE_PV_0
                x_1d_loc => x_1d(id); id = id+1
                x_1d_loc%var_name = 'RE_PV_0'
                allocate(x_1d_loc%tot_i_min(4),x_1d_loc%tot_i_max(4))
                allocate(x_1d_loc%loc_i_min(4),x_1d_loc%loc_i_max(4))
                x_1d_loc%tot_i_min = [1,1,1,1]
                x_1d_loc%tot_i_max = [n_tot,nn_mod_tot(1)]
                x_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,sxr_tot(1,1)]
                x_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,&
                    &sxr_tot(2,1)]
                allocate(x_1d_loc%p(loc_size(1)))
                x_1d_loc%p = reshape(rp(x%PV_0(par_id_loc(1):par_id_loc(2),:,&
                    &norm_id(1):norm_id(2),sxr_loc(1,1):sxr_loc(2,1))),&
                    &[loc_size(1)])

                ! IM_PV_0
                x_1d_loc => x_1d(id); id = id+1
                x_1d_loc%var_name = 'IM_PV_0'
                allocate(x_1d_loc%tot_i_min(4),x_1d_loc%tot_i_max(4))
                allocate(x_1d_loc%loc_i_min(4),x_1d_loc%loc_i_max(4))
                x_1d_loc%tot_i_min = [1,1,1,1]
                x_1d_loc%tot_i_max = [n_tot,nn_mod_tot(1)]
                x_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,sxr_tot(1,1)]
                x_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,&
                    &sxr_tot(2,1)]
                allocate(x_1d_loc%p(loc_size(1)))
                x_1d_loc%p = reshape(ip(x%PV_0(par_id_loc(1):par_id_loc(2),:,&
                    &norm_id(1):norm_id(2),sxr_loc(1,1):sxr_loc(2,1))),&
                    &[loc_size(1)])

                ! RE_PV_2
                x_1d_loc => x_1d(id); id = id+1
                x_1d_loc%var_name = 'RE_PV_2'
                allocate(x_1d_loc%tot_i_min(4),x_1d_loc%tot_i_max(4))
                allocate(x_1d_loc%loc_i_min(4),x_1d_loc%loc_i_max(4))
                x_1d_loc%tot_i_min = [1,1,1,1]
                x_1d_loc%tot_i_max = [n_tot,nn_mod_tot(1)]
                x_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,sxr_tot(1,1)]
                x_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,&
                    &sxr_tot(2,1)]
                allocate(x_1d_loc%p(loc_size(1)))
                x_1d_loc%p = reshape(rp(x%PV_2(par_id_loc(1):par_id_loc(2),:,&
                    &norm_id(1):norm_id(2),sxr_loc(1,1):sxr_loc(2,1))),&
                    &[loc_size(1)])

                ! IM_PV_2
                x_1d_loc => x_1d(id); id = id+1
                x_1d_loc%var_name = 'IM_PV_2'
                allocate(x_1d_loc%tot_i_min(4),x_1d_loc%tot_i_max(4))
                allocate(x_1d_loc%loc_i_min(4),x_1d_loc%loc_i_max(4))
                x_1d_loc%tot_i_min = [1,1,1,1]
                x_1d_loc%tot_i_max = [n_tot,nn_mod_tot(1)]
                x_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,sxr_tot(1,1)]
                x_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,&
                    &sxr_tot(2,1)]
                allocate(x_1d_loc%p(loc_size(1)))
                x_1d_loc%p = reshape(ip(x%PV_2(par_id_loc(1):par_id_loc(2),:,&
                    &norm_id(1):norm_id(2),sxr_loc(1,1):sxr_loc(2,1))),&
                    &[loc_size(1)])

                ! RE_KV_0
                x_1d_loc => x_1d(id); id = id+1
                x_1d_loc%var_name = 'RE_KV_0'
                allocate(x_1d_loc%tot_i_min(4),x_1d_loc%tot_i_max(4))
                allocate(x_1d_loc%loc_i_min(4),x_1d_loc%loc_i_max(4))
                x_1d_loc%tot_i_min = [1,1,1,1]
                x_1d_loc%tot_i_max = [n_tot,nn_mod_tot(1)]
                x_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,sxr_tot(1,1)]
                x_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,&
                    &sxr_tot(2,1)]
                allocate(x_1d_loc%p(loc_size(1)))
                x_1d_loc%p = reshape(rp(x%KV_0(par_id_loc(1):par_id_loc(2),:,&
```

```
                    &norm_id(1):norm_id(2),sxr_loc(1,1):sxr_loc(2,1))),&
                    &[loc_size(1)])

                ! IM_KV_0
                x_1d_loc => x_1d(id); id = id+1
                x_1d_loc%var_name = 'IM_KV_0'
                allocate(x_1d_loc%tot_i_min(4),x_1d_loc%tot_i_max(4))
                allocate(x_1d_loc%loc_i_min(4),x_1d_loc%loc_i_max(4))
                x_1d_loc%tot_i_min = [1,1,1,1]
                x_1d_loc%tot_i_max = [n_tot,nn_mod_tot(1)]
                x_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,sxr_tot(1,1)]
                x_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,&
                    &sxr_tot(2,1)]
                allocate(x_1d_loc%p(loc_size(1)))
                x_1d_loc%p = reshape(ip(x%KV_0(par_id_loc(1):par_id_loc(2),:,&
                    &norm_id(1):norm_id(2),sxr_loc(1,1):sxr_loc(2,1))),&
                    &[loc_size(1)])

                ! RE_KV_2
                x_1d_loc => x_1d(id); id = id+1
                x_1d_loc%var_name = 'RE_KV_2'
                allocate(x_1d_loc%tot_i_min(4),x_1d_loc%tot_i_max(4))
                allocate(x_1d_loc%loc_i_min(4),x_1d_loc%loc_i_max(4))
                x_1d_loc%tot_i_min = [1,1,1,1]
                x_1d_loc%tot_i_max = [n_tot,nn_mod_tot(1)]
                x_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,sxr_tot(1,1)]
                x_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,&
                    &sxr_tot(2,1)]
                allocate(x_1d_loc%p(loc_size(1)))
                x_1d_loc%p = reshape(rp(x%KV_2(par_id_loc(1):par_id_loc(2),:,&
                    &norm_id(1):norm_id(2),sxr_loc(1,1):sxr_loc(2,1))),&
                    &[loc_size(1)])

                ! IM_KV_2
                x_1d_loc => x_1d(id); id = id+1
                x_1d_loc%var_name = 'IM_KV_2'
                allocate(x_1d_loc%tot_i_min(4),x_1d_loc%tot_i_max(4))
                allocate(x_1d_loc%loc_i_min(4),x_1d_loc%loc_i_max(4))
                x_1d_loc%tot_i_min = [1,1,1,1]
                x_1d_loc%tot_i_max = [n_tot,nn_mod_tot(1)]
                x_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,sxr_tot(1,1)]
                x_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,&
                    &sxr_tot(2,1)]
                allocate(x_1d_loc%p(loc_size(1)))
                x_1d_loc%p = reshape(ip(x%KV_2(par_id_loc(1):par_id_loc(2),:,&
                    &norm_id(1):norm_id(2),sxr_loc(1,1):sxr_loc(2,1))),&
                    &[loc_size(1)])
            end if

            if (print_this(2)) then
                ! RE_PV_1
                x_1d_loc => x_1d(id); id = id+1
                x_1d_loc%var_name = 'RE_PV_1'
                allocate(x_1d_loc%tot_i_min(4),x_1d_loc%tot_i_max(4))
                allocate(x_1d_loc%loc_i_min(4),x_1d_loc%loc_i_max(4))
                x_1d_loc%tot_i_min = [1,1,1,1]
                x_1d_loc%tot_i_max = [n_tot,nn_mod_tot(2)]
                x_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,sxr_tot(1,2)]
                x_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,&
                    &sxr_tot(2,2)]
                allocate(x_1d_loc%p(loc_size(2)))
                x_1d_loc%p = reshape(rp(x%PV_1(par_id_loc(1):par_id_loc(2),:,&
                    &norm_id(1):norm_id(2),sxr_loc(1,2):sxr_loc(2,2))),&
                    &[loc_size(2)])

                ! IM_PV_1
                x_1d_loc => x_1d(id); id = id+1
                x_1d_loc%var_name = 'IM_PV_1'
                allocate(x_1d_loc%tot_i_min(4),x_1d_loc%tot_i_max(4))
                allocate(x_1d_loc%loc_i_min(4),x_1d_loc%loc_i_max(4))
                x_1d_loc%tot_i_min = [1,1,1,1]
                x_1d_loc%tot_i_max = [n_tot,nn_mod_tot(2)]
                x_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,sxr_tot(1,2)]
                x_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,&
                    &sxr_tot(2,2)]
                allocate(x_1d_loc%p(loc_size(2)))
                x_1d_loc%p = reshape(ip(x%PV_1(par_id_loc(1):par_id_loc(2),:,&
                    &norm_id(1):norm_id(2),sxr_loc(1,2):sxr_loc(2,2))),&
                    &[loc_size(2)])

                ! RE_KV_1
                x_1d_loc => x_1d(id); id = id+1
                x_1d_loc%var_name = 'RE_KV_1'
                allocate(x_1d_loc%tot_i_min(4),x_1d_loc%tot_i_max(4))
                allocate(x_1d_loc%loc_i_min(4),x_1d_loc%loc_i_max(4))
                x_1d_loc%tot_i_min = [1,1,1,1]
                x_1d_loc%tot_i_max = [n_tot,nn_mod_tot(2)]
```

```
                x_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,sxr_tot(1,2)]
                x_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,&
                    &sxr_tot(2,2)]
                allocate(x_1d_loc%p(loc_size(2)))
                x_1d_loc%p = reshape(rp(x%KV_1(par_id_loc(1):par_id_loc(2),:,&
                    &norm_id(1):norm_id(2),sxr_loc(1,2):sxr_loc(2,2))),&
                    &[loc_size(2)])

                ! IM_KV_1
                x_1d_loc => x_1d(id); id = id+1
                x_1d_loc%var_name = 'IM_KV_1'
                allocate(x_1d_loc%tot_i_min(4),x_1d_loc%tot_i_max(4))
                allocate(x_1d_loc%loc_i_min(4),x_1d_loc%loc_i_max(4))
                x_1d_loc%tot_i_min = [1,1,1,1]
                x_1d_loc%tot_i_max = [n_tot,nn_mod_tot(2)]
                x_1d_loc%loc_i_min = [par_id(1),1,grid_trim%i_min,sxr_tot(1,2)]
                x_1d_loc%loc_i_max = [par_id(2),n_tot(2),grid_trim%i_max,&
                    &sxr_tot(2,2)]
                allocate(x_1d_loc%p(loc_size(2)))
                x_1d_loc%p = reshape(ip(x%KV_1(par_id_loc(1):par_id_loc(2),:,&
                    &norm_id(1):norm_id(2),sxr_loc(1,2):sxr_loc(2,2))),&
                    &[loc_size(2)])
            end if
        end do

        ! write
        ierr = print_hdf5_arrs(x_1d(1:id-1),pb3d_name,trim(data_name),&
            &rich_lvl=rich_lvl,ind_print=.not.grid_trim%divided)
        chckerr('')

        ! clean up
        call grid_trim%dealloc()
        call dealloc_var_1d(x_1d)
        nullify(x_1d_loc)

        ! user output
        call lvl_ud(-1)
    end function print_output_x_2

    !> Initializes some variables concerning the mode numbers.
    !!
    !! Setup  minimum and  maximum  of mode  numbers at  every  flux surface  in
    !! equilibrium coordinates
    !!  - \c min_n_X, \c max_n_X
    !!  - \c min_m_X, \c max_m_X,
    !!
    !! It  functions depending  on  the \c  X_style used:  1  (prescribed) or  2
    !! (fast). For the fast  style, at every flux surface the  range of modes is
    !! sought that resonates most:
    !!  - \f$m = nq \pm \frac{N}{2}\f$      for poloidal flux
    !!  - \f$n = \iota m \pm \frac{N}{2}\f$  for toroidal flux
    !!
    !! where \f$N\f$ = \c n_mod_X.
    !!
    !! However, at  the same  time, both \c  m and  \c n have  to be  larger, in
    !! absolute  value, than  \c min_sec_X.  Therefore,  the range  of width  \c
    !! n_mod_X can be shifted upwards.
    !!
    !! \return ierr
    integer function init_modes(grid_eq,eq) result(ierr)
        use num_vars, only: use_pol_flux_f, x_style
        use x_vars, only: prim_x, min_sec_x, max_sec_x, n_mod_x, min_n_x, &
            &max_n_x, min_m_x, max_m_x, min_nm_x
        use mpi_utilities, only: get_ser_var
        use grid_utilities, only: trim_grid

        character(*), parameter :: rout_name = 'init_modes'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(eq_1_type), intent(in) :: eq                                       !< flux equilibrium

        ! local variables
        type(grid_type) :: grid_eq_trim                                         ! trimmed version of equilibrium
        real(dp), allocatable :: jq_tot(:)                                      ! saf. fac. or rot. transf. and derivs. in
Flux coords.
        integer :: norm_eq_id(2)                                                ! untrimmed normal indices for trimmed grids

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Initializing perturbation modes variables')
        call lvl_ud(1)

        ! get trimmed grids
        ierr = trim_grid(grid_eq,grid_eq_trim,norm_eq_id)
```

```
        chckerr('')

        ! (re)allocate variables
        if (allocated(min_n_x)) deallocate(min_n_x)
        if (allocated(max_n_x)) deallocate(max_n_x)
        if (allocated(min_m_x)) deallocate(min_m_x)
        if (allocated(max_m_x)) deallocate(max_m_x)
        allocate(min_n_x(grid_eq_trim%n(3)),max_n_x(grid_eq_trim%n(3)))
        allocate(min_m_x(grid_eq_trim%n(3)),max_m_x(grid_eq_trim%n(3)))

        ! get serial version of safety factor or rot. transform
        allocate(jq_tot(grid_eq_trim%n(3)))
        if (use_pol_flux_f) then
            if (grid_eq%divided) then
                ierr = get_ser_var(eq%q_saf_FD(norm_eq_id(1):norm_eq_id(2),0),&
                    &jq_tot,scatter=.true.)
                chckerr('')
            else
                jq_tot = eq%q_saf_FD(:,0)
            end if
        else
            if (grid_eq%divided) then
                ierr = get_ser_var(eq%rot_t_FD(norm_eq_id(1):norm_eq_id(2),0),&
                    &jq_tot,scatter=.true.)
                chckerr('')
            else
                jq_tot = eq%rot_t_FD(:,0)
            end if
        end if

        ! set the limits depending on the X style
        select case (x_style)
            case (1)                                                    ! prescribed
                if (use_pol_flux_f) then
                    min_n_x = prim_x
                    max_n_x = prim_x
                    min_m_x = min_sec_x
                    max_m_x = max_sec_x
                else
                    min_n_x = min_sec_x
                    max_n_x = max_sec_x
                    min_m_x = prim_x
                    max_m_x = prim_x
                end if
            case (2)                                                    ! fast
                if (use_pol_flux_f) then
                    min_n_x = prim_x
                    max_n_x = prim_x
                    if (prim_x*jq_tot(1).gt.0) then                     ! nq > 0: m > 0
                        min_m_x = nint(prim_x*jq_tot-(n_mod_x-1)*0.5)
                        min_m_x = max(min_nm_x,min_m_x)
                        max_m_x = min_m_x + n_mod_x - 1
                    else                                                ! nq < 0: m < 0
                        max_m_x = nint(prim_x*jq_tot+(n_mod_x-1)*0.5)
                        max_m_x = min(-min_nm_x,max_m_x)
                        min_m_x = max_m_x - n_mod_x + 1
                    end if
                else
                    if (prim_x*jq_tot(1).gt.0) then                     ! m iota > 0: n > 0
                        min_n_x = nint(prim_x*jq_tot-(n_mod_x-1)*0.5)
                        min_n_x = max(min_nm_x,min_n_x)
                        max_n_x = min_n_x + n_mod_x - 1
                    else                                                ! m iota < 0: n < 0
                        max_n_x = nint(prim_x*jq_tot+(n_mod_x-1)*0.5)
                        max_n_x = min(-min_nm_x,max_n_x)
                        min_n_x = max_n_x - n_mod_x + 1
                    end if
                    min_m_x = prim_x
                    max_m_x = prim_x
                end if
        end select

        ! clean up
        call grid_eq_trim%dealloc()

        call lvl_ud(-1)
        call writo('Perturbation modes variables initialized')
    end function init_modes

    !> Sets up some variables concerning the mode numbers.
    !!
    !! Apart from the mode numbers
    !!  - \c n, \c m,
    !!
    !! also the variables of the secondary modes
    !!  - sec,
    !!
```

```
    !! are set up  in the coordinates of  the grid passed. The  normal values of
    !! this grid are saved as well, in
    !!  - r_F.
    !!
    !! All variables  are tabulated in the  full normal grid. The  mode indices,
    !! however, are  chosen so that  there is maximum overlap  between different
    !! normal positions. This is trivial for \c  X_style 1 (prescribed), but for
    !! \c X_style 2 (fast), this is best explained by an example:
    !!
    !!  kd  m1 m2 m3
    !!  ------------
    !!   1  10 11 12 <- start
    !!   2  10 11 12 <- no change in limits
    !!   2  13 11 12 <- limits shift up by one
    !!   3  13 11 12 <- no change in limits
    !!   4  13 14 12 <- limits shift up by one
    !!   5  16 14 15 <- limits shift up by two
    !!   6  13 14 15 <- limits shift down by one
    !!
    !! This procedure makes use of the global variables
    !!  - \c min_m_X, max_m_X, min_n_X, max_m_X
    !!
    !! that have to be set up using init_nm_x().
    !!
    !! Optionally, \c n and \c m can be plot.
    !!
    !! \return ierr
    integer function setup_modes(mds,grid_eq,grid,plot_name) result(ierr)
        use num_vars, only: use_pol_flux_f, rank
        use x_vars, only: n_mod_x, min_n_x, min_m_x, max_n_x, max_m_x
        use grid_utilities, only: trim_grid
        use eq_vars, only: max_flux_f
        use num_utilities, only: spline

        character(*), parameter :: rout_name = 'setup_modes'

        ! input / output
        type(modes_type), intent(inout), target :: mds                          !< modes variables
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(grid_type), intent(in) :: grid                                     !< grid at which to calculate modes
        character(len=*), intent(in), optional :: plot_name                     !< name to be used when plotting \c n and \c m

        ! local variables
        real(dp), allocatable :: min_nm_x(:)                                    ! interpolated minimum mode number
        real(dp), allocatable :: x_plot(:,:)                                    ! abscissa of plot
        integer, pointer :: nm_x(:,:)                                           ! either n or m
        integer :: id, ld, kd                                                   ! counters
        integer :: ld_loc                                                       ! shfited ld
        integer :: n_mod_tot                                                    ! total number of modes
        integer :: ind_id                                                       ! current size of ind_tot
        integer :: ld_shift                                                     ! shift in table indices
        integer :: mod_x_range                                                  ! total mode range
        integer :: delta_ld                                                     ! change in total mode numbers
        integer, allocatable :: ind_cur(:)                                      ! current indices
        integer, allocatable :: ind_tot(:,:)                                    ! total index information, will be later cut to sec
        character(len=max_str_ln), allocatable :: plot_titles(:)                ! title for plots
        character(len=max_str_ln) :: plot_name_tot                              ! file name for plots
        character(len=max_str_ln) :: err_msg                                    ! error message
#if ldebug
        real(dp), allocatable :: y_plot(:,:)                                    ! ordinate of plot
#endif

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Setting up general modes variables')
        call lvl_ud(1)

        ! set up normal tabulation values
        allocate(ind_tot(-grid%n(3)*n_mod_x:grid%n(3)*n_mod_x,4))               ! not quite absolute maximum but should never be reached
        allocate(ind_cur(n_mod_x))                                              ! indices of total modes currently being treated

        ! calculate n and m
        allocate(mds%n(grid%n(3),n_mod_x))
        allocate(mds%m(grid%n(3),n_mod_x))

        ! auxiliary variable
        allocate(min_nm_x(grid%n(3)))

        ! iterate over n (id=1) and m (id=2)
        do id = 1,2
            ! point  nm_X, and  interpolate  mode minimum  (linear, to  preserve
```

```
            ! form)
            if (id.eq.1) then
                nm_x => mds%n
                mod_x_range = max_n_x(1)-min_n_x(1)                          ! assumes that this is true for all normal points
                ierr = spline(grid_eq%r_F,min_n_x*1._dp,grid%r_F,min_nm_x,ord=1)
                chckerr('')
            else
                nm_x => mds%m
                mod_x_range = max_m_x(1)-min_m_x(1)                          ! assumes that this is true for all normal points
                ierr = spline(grid_eq%r_F,min_m_x*1._dp,grid%r_F,min_nm_x,ord=1)
                chckerr('')
            end if

            !!! For non-monotonous tests:
            !!!if (id .eq. 2) then
                !!!min_nm_X(nint(grid%n(3)*0.7):grid%n(3)) = &
                    !!!&2*min_nm_X(nint(grid%n(3)*0.7)) - &
                    !!!&min_nm_X(nint(grid%n(3)*0.7):grid%n(3))
            !!!end if

            ! initialize variables for first normal grid point
            ld_shift = 0
            ind_id = 0
            do ld = 1,n_mod_x
                nm_x(1,ld) = nint(min_nm_x(1)) + mod_x_range*(ld-1)/(n_mod_x-1)
                ind_tot(ld,:) = [nm_x(1,ld),1,0,ld]
                ind_id = ind_id + 1
#if ldebug
                if (debug_setup_modes) write(*,*) 'starting with', ld, ':', &
                    &ind_tot(ld,:)
#endif
            end do
            ind_cur = [(ld, ld=1,n_mod_x)]

            ! iterate over all next normal grid points
            do kd = 2,grid%n(3)
                ! update ld delta and shift
                delta_ld = nint(min_nm_x(kd)) - nint(min_nm_x(kd-1))
                ld_shift = ld_shift + delta_ld

                ! set n or m
                do ld = 1,n_mod_x
                    ! shift ld and wrap back to [1..n_mod_X]
                    ld_loc = modulo(ld+ld_shift-1,n_mod_x)+1
                    nm_x(kd,ld_loc) = nint(min_nm_x(kd)) + &
                        &mod_x_range*(ld-1)/(n_mod_x-1)
                end do

                ! if there was a shift, set new total index information
                if (abs(delta_ld).gt.0) then
#if ldebug
                    if (debug_setup_modes) write(*,*) 'kd', kd, 'delta', &
                        &delta_ld
#endif
                    do ld = 1,abs(delta_ld)
                        if (delta_ld.gt.0) then                              ! mode number has increased
                            ind_tot(ind_cur(1),3) = kd - 1                   ! upper limit in normal range
                            ind_tot(ind_id+ld,:) = [ &
                                &ind_tot(ind_cur(n_mod_x),1) + 1, &          ! total mode number
                                &kd, &                                       ! lower limit in normal range
                                &0, &                                        ! initalize upper limit normal range
                                &modulo(ind_tot(ind_id,4)+ld-1,n_mod_x) + 1] ! index in tables, shifted and wrapped around
#if ldebug
                            if (debug_setup_modes) then
                                write(*,*) '    FINISHES', ind_cur(1), &
                                    &':', ind_tot(ind_cur(1),:)
                                write(*,*) '    STARTS', ind_id+ld, ':', &
                                    &ind_tot(ind_id+ld,:)
                            end if
#endif
                            ind_cur = [ind_cur(2:n_mod_x),ind_id+ld]         ! add ind_id+ld at top
                        else                                                 ! mode number has decreased
                            ind_tot(ind_cur(n_mod_x),3) = kd - 1             ! upper limit normal range
                            ind_tot(ind_id+ld,:) = [ &
                                &ind_tot(ind_cur(1),1) - 1, &                ! total mode number
                                &kd, &                                       ! lower limit in normal range
                                &0, &                                        ! initalize upper limit normal range
                                &modulo(ind_tot(ind_id,4)-ld,n_mod_x) + 1]   ! index in tables, shifted and wrapped around
#if ldebug
                            if (debug_setup_modes) then
                                write(*,*) '    FINISHES', ind_cur(n_mod_x), &
                                    &':', ind_tot(ind_cur(n_mod_x),:)
                                write(*,*) '    STARTS', ind_id+ld, ':', &
```

```
                                    &ind_tot(ind_id+ld,:)
                            end if
#endif
                            ind_cur = [ind_id+ld,ind_cur(1:n_mod_x-1)]          ! add ind_id+ld at bottom
                        end if
                    end do
                    ind_id = ind_id+abs(delta_ld)
#if ldebug
                    if (debug_setup_modes) write(*,*) '    current indices:', &
                        &ind_cur
#endif
                end if
            end do

            ! close last total modes
            do ld = 1,size(ind_cur)
                ind_tot(ind_cur(ld),3) = grid%n(3)
            end do

            ! save in index information
            if ((use_pol_flux_f .and. id.eq.2) .or. &
                &(.not.use_pol_flux_f .and. id.eq.1)) then
                allocate(mds%sec(ind_id,4))
                mds%sec = ind_tot(1:ind_id,:)
            end if
        end do

        ! master plots output if requested
        if (rank.eq.0 .and. present(plot_name)) then
            call writo('Plotting mode numbers')
            call lvl_ud(1)

            ! total number of modes
            n_mod_tot = size(mds%sec,1)

            ! set up x values
            allocate(x_plot(grid%n(3),n_mod_x))
            do ld = 1,n_mod_x
                x_plot(:,ld) = grid%r_F
            end do
            x_plot = x_plot*2*pi/max_flux_f
            allocate(plot_titles(1))

            ! plot poloidal modes
            plot_titles(1) = 'poloidal mode numbers'
            plot_name_tot = 'modes_m_'//trim(plot_name)
            call print_ex_2d(plot_titles(1:1),plot_name_tot,mds%m*1._dp,&
                &x=x_plot,draw=.false.)
            call draw_ex(plot_titles(1:1),plot_name_tot,n_mod_x,1,.false.)

            ! plot toroidal modes
            plot_titles(1) = 'toroidal mode numbers'
            plot_name_tot = 'modes_n_'//trim(plot_name)
            call print_ex_2d(plot_titles(1:1),plot_name_tot,mds%n*1._dp,&
                &x=x_plot,draw=.false.)
            call draw_ex(plot_titles(1:1),plot_name_tot,n_mod_x,1,.false.)

            ! clean up
            deallocate(x_plot)
            deallocate(plot_titles)

#if ldebug
            ! plot secondary limits
            allocate(x_plot(n_mod_tot,n_mod_tot+2))
            allocate(y_plot(n_mod_tot,n_mod_tot+2))
            allocate(plot_titles(n_mod_tot+2))
            do ld = 1,n_mod_tot
                x_plot(:,ld) = ld
                y_plot(:,ld) = mds%sec(ld,2)                                    ! lower boundary
                y_plot(n_mod_tot,ld) = mds%sec(ld,3)                            ! upper boundary
                plot_titles(ld) = 'mode '//trim(i2str(mds%sec(ld,1)))
            end do
            x_plot(:,n_mod_tot+1) = x_plot(1,1:n_mod_tot)
            y_plot(:,n_mod_tot+1) = mds%sec(:,1)                                ! mode number
            plot_titles(n_mod_tot+1) = 'mode number'
            x_plot(:,n_mod_tot+2) = x_plot(1,1:n_mod_tot)
            y_plot(:,n_mod_tot+2) = mds%sec(:,4)                                ! index
            plot_titles(n_mod_tot+2) = 'index'
            plot_name_tot = 'modes_sec_'//trim(plot_name)
            call print_ex_2d(plot_titles,plot_name_tot,y_plot,x=x_plot,&
                    &draw=.false.)
            call draw_ex(plot_titles,plot_name_tot,n_mod_tot+2,1,.false.)
            plot_name_tot = 'modes_sec_flux_'//trim(plot_name)
            do ld = 1,n_mod_tot
                y_plot(:,ld) = grid%r_F(mds%sec(ld,2))*2*pi/max_flux_f          ! lower boundary
                y_plot(n_mod_tot,ld) = grid%r_F(mds%sec(ld,3))*2*pi/max_flux_f  ! upper boundary
            end do
```

```
            call print_ex_2d(plot_titles(1:n_mod_tot),plot_name_tot,&
                &y_plot(:,1:n_mod_tot),x=x_plot(:,1:n_mod_tot),draw=.false.)
            call draw_ex(plot_titles(1:n_mod_tot),plot_name_tot,n_mod_tot,1,&
                &.false.)
            deallocate(x_plot)
            deallocate(y_plot)
            deallocate(plot_titles)
#endif

            call lvl_ud(-1)
        end if

        ! test whether all modes have at least one normal position
        if (minval(mds%sec(:,3)-mds%sec(:,2)+1).lt.1) then
            ierr = 1
            call writo('Mode '//&
                &trim(i2str(minloc(mds%sec(:,3)-mds%sec(:,2),1)))//&
                &' is not present in any flux surface')
            err_msg = 'Aument number of modes or choose finer equilibrium'
            chckerr(err_msg)
        end if

        ! clean up
        nullify(nm_x)

        call lvl_ud(-1)
        call writo('General mode variables set up')
    end function setup_modes

    !> Checks whether the high-n approximation is valid:
    !!
    !! This depends on the \c X_style:
    !!
    !! For X_style 1 (prescribed): every mode should resonate at least somewhere
    !! in the whole normal range:
    !!  - \f$\frac{\left|nq-m\right|}{\left|n\right|} < T\f$ and
    !!      \f$\frac{\left|nq-m\right|}{\left|m\right|} < T\f$
    !!      for poloidal flux
    !!  - \f$\frac{\left|q-\iota m\right|}{\left|m\right|} < T\f$ and
    !!      \f$\frac{\left|n-\iota m\right|}{\left|n\right|} < T\f$
    !!      for toroidal flux
    !!
    !! where \f$T\f$ = \c tol << 1.
    !!
    !! This condition is determined by the sign of \f$q\f$ (or \f$\iota\f$) and
    !! given by:
    !!  - \f$\max\left(q_\text{min}-T,\frac{q_\text{min}}{1+T}\right) <
    !!      \frac{m}{n} <
    !!      \min\left(q_\text{max}+T,\frac{q_\text{max}}{1-T}\right)\f$,
    !!      \f$q>0\f$
    !!  - \f$\max\left(q_\text{min}-T,\frac{q_\text{min}}{1-T}\right) <
    !!      \frac{m}{n} <
    !!      \min\left(q_\text{max}+T,\frac{q_\text{max}}{1+T}\right)\f$,
    !!      \f$q<0\f$
    !!
    !! for poloidal flux.
    !!
    !! For  toroidal  flux,  \f$q\f$  should  be  replaced  by  \f$\iota\f$  and
    !! \f$\frac{m}{n}\f$ by \f$\frac{n}{m}\f$.
    !!
    !! For \c  X_style 2 (fast):  the resonance has been  taken care of,  but it
    !! remains to be checked whether the number of modes is efficient.
    !!
    !! \return ierr
    integer function check_x_modes(grid_eq,eq) result(ierr)
        use num_vars, only: x_style, rank, use_pol_flux_f, n_procs
        use mpi_utilities, only: get_ser_var

        character(*), parameter :: rout_name = 'check_X_modes'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                              !< equilibrium grid
        type(eq_1_type), intent(in) :: eq                                   !< flux equilibrium

        ! local variables
        character(len=max_str_ln) :: err_msg                                ! error message

        ! initialize ierr
        ierr = 0

        call writo('Checking mode numbers')
        call lvl_ud(1)

        ! check the modes depending on the X style
        select case (x_style)
            case (1)                                                        ! prescribed
                ierr = check_x_modes_1(eq)
```

```
                chckerr('')
            case (2)                                                    ! fast
                ierr = check_x_modes_2(grid_eq,eq)
                chckerr('')
        end select

        call lvl_ud(-1)
        call writo('Mode numbers checked')
    contains
        ! Version for  X style 1: Checks  whether there exists a  range in which
        ! each of the modes resonates, with a certain tolerance tol_norm.
        !> \private
        integer function check_x_modes_1(eq) result(ierr)
            use x_vars, only: prim_x, min_sec_x, n_mod_x
            use num_vars, only: tol_norm

            character(*), parameter :: rout_name = 'check_X_modes_1'

            ! input / output
            type(eq_1_type), intent(in) :: eq                           ! flux equilibrium

            ! local variables
            integer :: id                                               ! counter
            integer :: pmone                                            ! plus or minus one
            real(dp) :: min_jq, max_jq                                  ! min. and max. values of q or iota
            real(dp) :: lim_lo, lim_hi                                  ! lower and upper limit on n/m (or m/n)
            real(dp), allocatable :: temp_var(:)                        ! temporary variable
            character(len=max_str_ln) :: jq_name                        ! either safety factor or rotational transform
            character(len=1) :: mode_name                               ! either n or m
            logical :: all_modes_fine                                   ! all modes are fine

            ! initialize ierr
            ierr = 0

            ! user output
            call writo('The tolerance used is '//trim(r2strt(tol_norm))//'...')

            ! set min_jq and max_jq in Flux coordinate system
            if (use_pol_flux_f) then
                min_jq = minval(eq%q_saf_FD(:,0))
                max_jq = maxval(eq%q_saf_FD(:,0))
            else
                min_jq = minval(eq%rot_t_FD(:,0))
                max_jq = maxval(eq%rot_t_FD(:,0))
            end if

            ! combine all processes
            ierr = get_ser_var([min_jq],temp_var)
            chckerr('')
            if (rank.eq.0) min_jq = minval(temp_var)
            ierr = get_ser_var([max_jq],temp_var)
            chckerr('')
            if (rank.eq.0) max_jq = maxval(temp_var)

            ! output if master
            if (rank.eq.0) then
                ! set up jq name and all_modes_fine
                if (use_pol_flux_f) then
                    jq_name = 'safety factor'
                    mode_name = 'm'
                else
                    jq_name = 'rotational transform'
                    mode_name = 'n'
                end if
                all_modes_fine = .true.

                ! set up plus minus one, according to the sign of jq
                if (min_jq.lt.0 .and. max_jq.lt.0) then
                    pmone = -1
                else if (min_jq.gt.0 .and. max_jq.gt.0) then
                    pmone = 1
                else
                    err_msg = trim(jq_name)//' cannot change sign'
                    ierr = 1
                    chckerr(err_msg)
                end if

                ! calculate upper and lower limits
                lim_lo = max(min_jq-tol_norm,min_jq/(1+pmone*tol_norm))
                lim_hi = min(max_jq+tol_norm,max_jq/(1-pmone*tol_norm))

                ! for every mode (n,m) check whether  m/n is inside the range of
                ! q values or n/m inside the range of iota values
                do id = 1,n_mod_x
                    ! check if limits are met
                    if (use_pol_flux_f) then
```

```
                        if ((min_sec_x+id-1._dp)/prim_x.lt.lim_lo .or. &
                            &(min_sec_x+id-1._dp)/prim_x.gt.lim_hi) then
                            call writo('for (n,m) = ('//trim(i2str(prim_x))//&
                                &','//trim(i2str(min_sec_x+id-1))//&
                                &'), there is no range in the plasma where the &
                                &ratio |n q - m| << |n|,|m| is met',alert=.true.)
                            all_modes_fine = .false.
                        end if
                    else
                        if ((min_sec_x+id-1._dp)/prim_x.lt.lim_lo .or. &
                            &(min_sec_x+id-1._dp)/prim_x.gt.lim_hi) then
                            call writo('for (n,m) = ('//&
                                &trim(i2str(min_sec_x+id-1))//','//&
                                &trim(i2str(prim_x))//'), there is no range in &
                                &the plasma where the ratio |n - iota m| << &
                                &|m|,|n| is met',alert=.true.)
                            all_modes_fine = .false.
                        end if
                    end if
                end do

                ! output message
                if (all_modes_fine) then
                    call writo('The modes are all within the allowed range &
                        &of '//trim(i2str(ceiling(prim_x*lim_lo)))//' < '//&
                        &mode_name//' < '//trim(i2str(floor(prim_x*lim_hi)))//&
                        &'...')
                else
                    call writo('Not all modes are within the allowed range &
                        &of '//trim(i2str(ceiling(prim_x*lim_lo)))//' < '//&
                        &mode_name//' < '//trim(i2str(floor(prim_x*lim_hi)))//&
                        &'...')
                end if
            end if
        end function check_x_modes_1

        ! version for X style 2: Check  how efficient the chosen number of modes
        ! is.
        !> \private
        integer function check_x_modes_2(grid_eq,eq) result(ierr)
            use x_vars, only: min_n_x, min_m_x, n_mod_x
            use grid_utilities, only: trim_grid

            character(*), parameter :: rout_name = 'check_X_modes_2'

            ! input / output
            type(grid_type), intent(in) :: grid_eq                          ! equilibrium grid
            type(eq_1_type), intent(in) :: eq                               ! flux equilibrium

            ! local variables
            type(grid_type) :: grid_eq_trim                                 ! trimmed equilibrium grid
            integer :: norm_id(2)                                           ! untrimmed normal indices for trimmed grid
            integer :: jd, kd, ld                                           ! counters
            real(dp), allocatable :: max_frac(:,:)                          ! maximum fraction
            real(dp), allocatable :: max_frac_sum(:)                        ! sum of maximum fraction for all processes
            real(dp), allocatable :: max_frac_max(:)                        ! maximum maximum fraction for all processes
            real(dp), allocatable :: max_frac_temp(:)                       ! auxilliary variable
            real(dp), allocatable :: n(:,:), m(:,:)                         ! n and m
            real(dp), allocatable :: fac_n(:), fac_m(:)                     ! factors to be multiplied with n m m
            character(len=max_str_ln) :: frac_name                          ! name of fraction
#if ldebug
            real(dp), allocatable :: x_plot(:,:)                            ! for plotting
            character(len=max_str_ln) :: plot_title                         ! title for plots
            character(len=max_str_ln) :: plot_name                          ! file name for plots
#endif

            ! initialize ierr
            ierr = 0

            ! set up helper variables
            allocate(max_frac(grid_eq%loc_n_r,n_mod_x))
            allocate(fac_n(grid_eq%loc_n_r))
            allocate(fac_m(grid_eq%loc_n_r))
            allocate(n(grid_eq%loc_n_r,n_mod_x))
            allocate(m(grid_eq%loc_n_r,n_mod_x))
            if (use_pol_flux_f) then
                do jd = 1,n_mod_x
                    n(:,jd) = min_n_x(grid_eq%i_min:grid_eq%i_max)
                    m(:,jd) = min_m_x(grid_eq%i_min:grid_eq%i_max)+jd-1
                end do
                fac_n = eq%q_saf_FD(:,0)
                fac_m = 1._dp
            else
                do jd = 1,n_mod_x
                    n(:,jd) = min_n_x(grid_eq%i_min:grid_eq%i_max)+jd-1
                    m(:,jd) = min_m_x(grid_eq%i_min:grid_eq%i_max)
                end do
```

```
                fac_n = 1._dp
                fac_m = eq%rot_t_FD(:,0)
            end if

            ! loop over all modes
            do ld = 1,n_mod_x
                do kd = 1,grid_eq%loc_n_r
                    max_frac(kd,ld) = &
                        &abs(n(kd,ld)*fac_n(kd)-m(kd,ld)*fac_m(kd))/&
                        &min(abs(n(kd,ld)),abs(m(kd,ld)))
                end do
            end do

            ! trim grid
            ierr = trim_grid(grid_eq,grid_eq_trim,norm_id=norm_id)
            chckerr('')

            ! gather all fractions
            if (rank.eq.0) allocate(max_frac_temp(n_procs))
            if (rank.eq.0) allocate(max_frac_sum(n_mod_x))
            if (rank.eq.0) allocate(max_frac_max(n_mod_x))
            do ld = 1,n_mod_x
                ierr = get_ser_var([sum(max_frac(norm_id(1):norm_id(2),ld))],&
                    &max_frac_temp)
                chckerr('')
                if (rank.eq.0) max_frac_sum(ld) = sum(max_frac_temp)
                ierr = get_ser_var(&
                    &[maxval(max_frac(norm_id(1):norm_id(2),ld))],max_frac_temp)
                chckerr('')
                if (rank.eq.0) max_frac_max(ld) = maxval(max_frac_temp)
            end do

            ! output if master
            if (rank.eq.0) then
                if (use_pol_flux_f) then
                    frac_name = '|nq-m|/|n| or |nq-m|/|m|'
                else
                    frac_name = '|n-iota m|/|n| or |n-iota m|/|m|'
                end if
                call writo('The fraction '//trim(frac_name)//' is')
                call lvl_ud(1)
                do ld = 1,n_mod_x
                    call writo('for mode '//trim(i2str(ld))//' maximally '//&
                        &trim(r2strt(max_frac_max(ld)))//&
                        &', average '//&
                        &trim(r2strt(max_frac_sum(ld)/grid_eq%n(3))))
                end do
                if (n_mod_x.gt.1) call writo('so for all modes maximally '//&
                    &trim(r2strt(maxval(max_frac_max)))//', average '//&
                    &trim(r2strt(sum(max_frac_sum)/(grid_eq%n(3)*n_mod_x))))
                call lvl_ud(-1)
            end if
#if ldebug
            if (debug_check_x_modes_2) then
                call writo('Plotting the fraction for all modes')
                allocate(x_plot(grid_eq%n(3),n_mod_x))
                do ld = 1,n_mod_x
                    x_plot(:,ld) = grid_eq%r_F
                end do
                plot_name = 'TEST_max_frac'
                plot_title = 'maximum fraction'
                call print_ex_2d([plot_title],plot_name,max_frac,x=x_plot,&
                    &draw=.false.)
                call draw_ex([plot_title],plot_name,n_mod_x,1,.false.)
            end if
#endif

            ! clean up
            call grid_eq_trim%dealloc()
        end function check_x_modes_2
    end function check_x_modes

    !> Calculates resonating flux surfaces for the perturbation modes.
    !!
    !! The output  consists of mode  number, resonating normal position  and the
    !! fraction  \f$\frac{m}{n}\f$ or  \f$\frac{n}{m}\f$.  for  those modes  for
    !! which a solution is found that is within the plasma range.
    !!
    !! It contains three pieces of information:
    !!  - <tt>(:,1)</tt>: the mode index
    !!  - <tt>(:,2)</tt>: the radial position in Flux coordinates
    !!  - <tt>(:,3)</tt>: the fraction \f$\frac{m}{n}\f$ or  \f$\frac{n}{m}\f$
    !!
    !! for every single mode  in \c sec of \c mds, which can  be tabulated in an
    !! arbitrary grid, not necessarily the equilibrium one.
    !!
    !! Optionally,  the  total safety  factor  or  rotational transform  can  be
```

```
    !! returned to the master.
    !!
    !! Also, information can be displayed with info.
    !!
    !! \return ierr
    integer function calc_res_surf(mds,grid_eq,eq,res_surf,info,jq) result(ierr)
        use x_vars, only: prim_x
        use num_vars, only: use_pol_flux_f, norm_disc_prec_eq
        use eq_vars, only: max_flux_f, max_flux_f
        use num_ops, only: calc_zero_zhang
        use grid_utilities, only: trim_grid
        use mpi_utilities, only: get_ser_var

        character(*), parameter :: rout_name = 'calc_res_surf'

        ! input / output
        type(modes_type), intent(in) :: mds                                    !< general modes variables
        type(grid_type), intent(in) :: grid_eq                                 !< equilibrium grid_eq
        type(eq_1_type), intent(in) :: eq                                      !< flux equilibrium
        real(dp), intent(inout), allocatable :: res_surf(:,:)                  !< resonant surface
        logical, intent(in), optional :: info                                  !< if info is displayed
        real(dp), intent(inout), optional, allocatable :: jq(:)                !< either safety factor or rotational transform

        ! local variables (also used in child functions)
        real(dp) :: nmfrac_fun                                                 ! fraction m/n or n/m to determine resonant flux surface

        ! local variables (not to be used in child functions)
        integer :: norm_id(2)                                                  ! untrimmed normal indices for trimmed grid
        integer :: ld                                                          ! counter
        integer :: ld_loc                                                      ! local ld
        logical :: info_loc                                                    ! local version of info
        real(dp), allocatable :: res_surf_loc(:,:)                             ! local copy of res_surf
        real(dp), allocatable :: jq_tot(:)                                     ! saf. fac. or rot. transf. and derivs. in Flux coords.
        real(dp), allocatable :: jq_loc(:)                                     ! local version of jq
        real(dp) :: norm_factor                                                ! normalization factor to make normal coordinate 0..1
        type(grid_type) :: grid_eq_trim                                        ! trimmed version of equilibrium grid
        integer :: n_loc, m_loc                                                ! local n and m
        character(len=max_str_ln) :: err_msg                                   ! error message

        ! initialize ierr
        ierr = 0

        ! set local info
        info_loc = .false.
        if (present(info)) info_loc = info

        ! get trimmed grid
        ierr = trim_grid(grid_eq,grid_eq_trim,norm_id)
        chckerr('')

        ! get serial version of safety factor or rot. transform
        allocate(jq_tot(grid_eq_trim%n(3)))
        if (use_pol_flux_f) then
            if (grid_eq%divided) then
                ierr = get_ser_var(eq%q_saf_FD(norm_id(1):norm_id(2),0),&
                    &jq_loc,scatter=.true.)
                chckerr('')
                jq_tot = jq_loc
            else
                jq_tot = eq%q_saf_FD(:,0)
            end if
        else
            if (grid_eq%divided) then
                ierr = get_ser_var(eq%rot_t_FD(norm_id(1):norm_id(2),0),&
                    &jq_loc,scatter=.true.)
                chckerr('')
                jq_tot = jq_loc
            else
                jq_tot = eq%rot_t_FD(:,0)
            end if
        end if
        if (present(jq)) then
            allocate(jq(grid_eq_trim%n(3)))
            jq = jq_tot
        end if

        ! initialize local res_surf
        allocate(res_surf_loc(1:size(mds%sec,1),3))

        ! calculate normalization factor max_flux / 2pi
        norm_factor = max_flux_f/(2*pi)

        ! loop over all modes (and shift the index in m_loc and n_loc by 1)
```

```
        ld_loc = 1
        do ld = 1,size(mds%sec,1)
            ! Find place  where q  = m/n or  iota = n/m  in Flux  coordinates by
            ! solving q-m/n = 0 or iota-n/m=0, using the functin jq_fun.

            ! set up local m, n and nmfrac for function
            if (use_pol_flux_f) then
                n_loc = prim_x
                m_loc = mds%sec(ld,1)
                nmfrac_fun = 1.0_dp*m_loc/n_loc
            else
                n_loc = mds%sec(ld,1)
                m_loc = prim_x
                nmfrac_fun = 1.0_dp*n_loc/m_loc
            end if

            ! calculate zero using Zhang
            if (use_pol_flux_f) then
                res_surf_loc(ld_loc,1) = m_loc
            else
                res_surf_loc(ld_loc,1) = n_loc
            end if
            res_surf_loc(ld_loc,3) = nmfrac_fun
            err_msg = calc_zero_zhang(res_surf_loc(ld_loc,2),jq_fun,&
                &[minval(grid_eq_trim%r_F),maxval(grid_eq_trim%r_F)])

            ! intercept error
            if (err_msg.ne.'') then
                if (info_loc) then
                    call writo('Mode (n,m) = ('//trim(i2str(n_loc))//','//&
                        &trim(i2str(m_loc))//') is not found')
                    call lvl_ud(1)
                    call writo(trim(err_msg))
                    call lvl_ud(-1)
                end if
            else if (res_surf_loc(ld_loc,2).lt.minval(grid_eq_trim%r_F) .or. &
                &res_surf_loc(ld_loc,2).gt.maxval(grid_eq_trim%r_F)) then
                if (info_loc) call writo('Mode (n,m) = ('//&
                    &trim(i2str(n_loc))//','//trim(i2str(m_loc))//&
                    &') does not resonate in plasma')
            else
                if (info_loc) call writo('Mode (n,m) = ('//&
                    &trim(i2str(n_loc))//','//trim(i2str(m_loc))//&
                    &') resonates in plasma at normalized flux surface '//&
                    &trim(r2str(res_surf_loc(ld_loc,2)/norm_factor)))
                ld_loc = ld_loc + 1                                     ! advance ld_loc
            end if
        end do

        ! set res_surf from local copy
        allocate(res_surf(ld_loc-1,3))
        res_surf = res_surf_loc(1:ld_loc-1,:)

        ! clean up
        call grid_eq_trim%dealloc()
    contains
        ! Returns q-m/n or  iota-n/m in Flux coordinates, used to  solve for q =
        ! m/n or iota = n/m.
        !> \private
        real(dp) function jq_fun(pt) result(res)
            use num_utilities, only: spline

            ! input / output
            real(dp), intent(in) :: pt                                  ! normal position at which to evaluate

            ! local variables
            integer :: i_min, i_max                                     ! index of minimum and maximum value of x
            real(dp) :: x_min, x_max                                    ! minimum and maximum value of x
            real(dp) :: res_loc(1)                                      ! local copy of res

            ! initialize res
            res = 0

            ! set up min. and max index and value
            x_min = minval(grid_eq_trim%r_F)
            x_max = maxval(grid_eq_trim%r_F)
            i_min = minloc(grid_eq_trim%r_F,1)
            i_max = maxloc(grid_eq_trim%r_F,1)

            ! check whether to interpolate or extrapolate
            if (pt.ge.x_min .and. pt.le.x_max) then                     ! point requested within range
                ! interpolate
                ierr = spline(grid_eq_trim%r_F,jq_tot-nmfrac_fun,[pt],res_loc,&
                    &ord=norm_disc_prec_eq)
                chckerr('')
                ! copy
                res = res_loc(1)
```

```
            end if
        end function jq_fun
    end function calc_res_surf

    !> plot  \f$q\f$-profile or  \f$\iota\f$-profile  in  flux coordinates  with
    !! \f$nq-m = 0\f$ or \f$n-\iota m = 0\f$ indicated if requested.
    !!
    !! The plot will be done in the grid in which \c mds is tabulated, which is
    !! not necessarily the equilibrium one.
    !!
    !! \return ierr
    integer function resonance_plot(mds,grid_eq,eq) result(ierr)
        use num_vars, only: use_pol_flux_f, no_plots, n_theta_plot, &
            &n_zeta_plot, rank, eq_style, min_theta_plot, max_theta_plot, &
            &min_zeta_plot, max_zeta_plot
        use eq_vars, only: max_flux_f
        use grid_utilities, only: calc_xyz_grid, calc_eqd_grid, coord_f2e, &
            &trim_grid
        use x_vars, only: prim_x
        use vmec_utilities, only: calc_trigon_factors

        character(*), parameter :: rout_name = 'resonance_plot'

        ! input / output
        type(modes_type), intent(in) :: mds                                     !< general modes variables
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(eq_1_type), intent(in) :: eq                                       !< flux equilibrium

        ! local variables (not to be used in child functions)
        integer :: ld                                                           ! counter
        integer :: n_mod_loc                                                    ! local n_mod
        real(dp), allocatable :: res_surf(:,:)                                  ! resonant surfaces
        real(dp), allocatable :: x_plot_loc(:,:)                                ! for plotting
        real(dp), allocatable :: y_plot_loc(:,:)                                ! for plotting
        character(len=max_str_ln) :: plot_title, file_name                      ! name of plot, of file
        real(dp), allocatable :: jq(:)                                          ! saf. fac. or rot. transf. in Flux coords.
        integer :: n_r                                                          ! total number of normal points
        integer :: plot_dim(4)                                                  ! plot dimensions (total = local because only masters)
        type(grid_type) :: grid_trim                                            ! trimmed version of grid
        type(grid_type) :: grid_plot                                            ! grid for plotting
        real(dp), allocatable :: r_plot_e(:)                                    ! normal E coordinates of plot (needed to calculate X, Y and Z)
        real(dp), allocatable :: theta_plot(:,:,:), zeta_plot(:,:,:)            ! pol. and tor. angle of plot
        real(dp), allocatable :: x_plot(:,:,:,:), y_plot(:,:,:,:), &
            &Z_plot(:,:,:,:)                                                    ! X, Y and Z of plot of all surfaces
        real(dp), allocatable :: vars(:,:,:,:)                                  ! variable to plot
        character(len=max_str_ln), allocatable :: plot_titles(:)                ! name of plots

        ! initialize ierr
        ierr = 0

        ! bypass plots if no_plots
        if (no_plots) return

        ! get trimmed grid
        ierr = trim_grid(grid_eq,grid_trim)
        chckerr('')

        ! initialize variables
        n_r = grid_trim%n(3)

        ! print user output
        if (use_pol_flux_f) then
            call writo('Plotting safety factor q and resonant surfaces &
                &q = m/n...')
            plot_title = 'safety factor q'
            file_name = 'q_saf'
        else
            call writo('Plotting rotational transform iota and resonant &
                &surfaces iota = n/m...')
            plot_title = 'rotational transform iota'
            file_name = 'rot_t'
        end if

        call lvl_ud(1)

        call writo('Calculate resonant surfaces')
        call lvl_ud(1)

        ! find resonating surfaces
        ierr = calc_res_surf(mds,grid_eq,eq,res_surf,info=.true.,jq=jq)
        chckerr('')

        call lvl_ud(-1)

        ! only master
```

```fortran
        if (rank.eq.0) then
            ! set local n_mod
            n_mod_loc = size(res_surf,1)

            ! set up plot titles
            allocate(plot_titles(n_mod_loc))
            if (use_pol_flux_f) then                                        ! n is fixed and m = m/n n
                do ld = 1,n_mod_loc
                    plot_titles(ld) = 'resonance for m,n = '//&
                        &trim(i2str(nint(res_surf(ld,3)*prim_x)))//','//&
                        &trim(i2str(prim_x))
                end do
            else                                                            ! m is fixed and n = n/m m
                do ld = 1,n_mod_loc
                    plot_titles(ld) = 'resonance for m,n = '//&
                        &trim(i2str(prim_x))//','//&
                        &trim(i2str(nint(res_surf(ld,3)*prim_x)))
                end do
            end if

            ! initialize x_plot_loc and y_plot_loc
            allocate(x_plot_loc(n_r,n_mod_loc+1)); x_plot_loc = 0
            allocate(y_plot_loc(n_r,n_mod_loc+1)); y_plot_loc = 0

            ! set x_plot_loc and y_plot_loc for first column
            x_plot_loc(:,1) = grid_trim%r_F
            y_plot_loc(:,1) = jq(:)

            ! set x_plot_loc and y_plot_loc for other columns
            do ld = 1,n_mod_loc
                x_plot_loc(:,ld+1) = res_surf(ld,2)
                y_plot_loc(n_r,ld+1) = res_surf(ld,3)
            end do

            ! only if resonant surfaces
            if (size(res_surf,1).gt.0) then
                ! user message
                call writo('Plot resonant flux surfaces using HDF5')

                call lvl_ud(1)

                ! set up pol. and tor. angle for plot
                allocate(theta_plot(n_theta_plot,n_zeta_plot,1))
                allocate(zeta_plot(n_theta_plot,n_zeta_plot,1))
                ierr = calc_eqd_grid(theta_plot,min_theta_plot*pi,&
                    &max_theta_plot*pi,1)
                chckerr('')
                ierr = calc_eqd_grid(zeta_plot,min_zeta_plot*pi,&
                    &max_zeta_plot*pi,2)
                chckerr('')

                ! set up vars
                allocate(vars(n_theta_plot,n_zeta_plot,1,n_mod_loc))
                do ld = 1,n_mod_loc
                    vars(:,:,:,ld) = y_plot_loc(n_r,ld+1)
                end do

                ! set dimensions for single flux surface
                plot_dim = [n_theta_plot,n_zeta_plot,1,n_mod_loc]

                ! calculate normal vars in Equilibrium coords.
                allocate(r_plot_e(n_mod_loc))
                ierr = coord_f2e(grid_eq,x_plot_loc(n_r,2:n_mod_loc+1),&
                    &r_plot_e,r_f_array=grid_eq%r_F,r_e_array=grid_eq%r_E)
                chckerr('')

                ! create plot grid for single flux surface
                ierr = grid_plot%init(plot_dim(1:3))
                chckerr('')
                grid_plot%theta_E = theta_plot
                grid_plot%zeta_E = zeta_plot

                ! if VMEC, calculate trigonometric factors of plot grid
                if (eq_style.eq.1) then
                    ierr = calc_trigon_factors(grid_plot%theta_E,&
                        &grid_plot%zeta_E,grid_plot%trigon_factors)
                    chckerr('')
                end if

                ! set up plot X, Y and Z
                allocate(x_plot(n_theta_plot,n_zeta_plot,1,n_mod_loc))
                allocate(y_plot(n_theta_plot,n_zeta_plot,1,n_mod_loc))
                allocate(z_plot(n_theta_plot,n_zeta_plot,1,n_mod_loc))

                ! loop over all resonant surfaces to calculate X, Y and Z values
                do ld = 1,n_mod_loc
                    ! set loc_r_E of plot grid
```

```
                    grid_plot%loc_r_E = r_plot_e(ld)

                    ! calculate X, Y and Z
                    ierr = calc_xyz_grid(grid_eq,grid_plot,x_plot(:,:,:,ld),&
                        &y_plot(:,:,:,ld),z_plot(:,:,:,ld))
                    chckerr('')
                end do

                ! plot using HDF5
                call plot_hdf5(plot_titles,file_name,vars,x=x_plot,y=y_plot,&
                    &z=z_plot,col=1,descr='resonant surfaces')

                ! deallocate local variables
                deallocate(vars)
                deallocate(theta_plot,zeta_plot,r_plot_e)
                deallocate(x_plot,y_plot,z_plot)
                call grid_plot%dealloc()

                call lvl_ud(-1)
            end if

            call writo('Plot safety factor with resonant flux surfaces using &
                &external program')

            call lvl_ud(1)

            ! rescale x_plot_loc by max_flux_F/2*pi
            x_plot_loc = x_plot_loc*2*pi/max_flux_f

            ! plot using external program
            call print_ex_2d([plot_title,plot_titles],file_name,y_plot_loc,&
                &x=x_plot_loc,draw=.false.)
            call draw_ex([plot_title,plot_titles],file_name,n_mod_loc+1,1,&
                &.false.)

            call lvl_ud(-1)

            ! only if resonant surfaces
            if (size(res_surf,1).gt.0) then
                call writo('Plot 2D map of resonances using HDF5')

                call lvl_ud(1)

                ! initialize plot dimensions
                plot_dim = [1,n_mod_loc,1,0]

                ! set up X, Y and Z
                allocate(x_plot(plot_dim(1),plot_dim(2),plot_dim(3),1))
                allocate(y_plot(plot_dim(1),plot_dim(2),plot_dim(3),1))
                allocate(z_plot(plot_dim(1),plot_dim(2),plot_dim(3),1))

                do ld = 1,n_mod_loc
                    x_plot(1,ld,1,1) = x_plot_loc(1,ld+1)
                    y_plot(1,ld,1,1) = res_surf(ld,1)
                    z_plot(1,ld,1,1) = 1._dp
                end do

                ! plot 2D (to be used with plots of harmonics in sol_ops)
                call plot_hdf5('resonating surfaces','res_surf',&
                    &z_plot(:,:,:,1),x=x_plot(:,:,:,1),y=y_plot(:,:,:,1))

                ! deallocate local variables
                deallocate(x_plot,y_plot,z_plot)

                call lvl_ud(-1)
            end if
        end if

        ! clean up
        call grid_trim%dealloc()

        call lvl_ud(-1)
    end function resonance_plot

    !> Calculate \f$U_m^0\f$, \f$U_m^1\f$ or \f$U_n^0\f$, \f$U_n^1\f$.
    !!
    !! This is done at  eq \c loc_n_r values of the  normal coordinate, \c n_par
    !! values of  the parallel coordinate and  \c size_X values of  the poloidal
    !! mode number,  or of  the toroidal  mode number, as  well as  the poloidal
    !! derivatives
    !!
    !! Use is made  of variables \c Ti  that are set up in  the equilibrium grid
    !! and are afterwards  converted to \c Ti_X in the  perturbation grid, which
    !! needs to have the same angular coordinates as the equilibrium grid:
    !!  \f[\begin{aligned}
    !!      T_1 &= \frac{B_\alpha}{B_\theta} \\
    !!      T_2 &= \Theta^\alpha + q' \theta \\
```

```
    !!      T_3 &= \frac{B_\alpha}{B_\theta} q' +
    !!          \frac{\mathcal{J}^2}{B_\theta} \mu_0 p' \\
    !!      T_4 &= \frac{B_\alpha}{B_\theta} q' \theta -
    !!          \frac{B_\psi}{B_\theta} \\
    !!      T_5 &= \frac{B_\alpha}{B_\theta}
    !!          \frac{\partial \Theta^\theta}{\partial \theta} -
    !!          \frac{\partial\Theta^\theta}{\partial \alpha} \\
    !!      T_6 &= \frac{B_\alpha}{B_\theta} \Theta^\theta \ ,
    !!  \end{aligned}\f]
    !! with \f$T_i\f$ = \c Ti.
    !!
    !! which is valid for poloidal Flux coordinates.
    !!
    !! For  toroidal  Flux  coordinates,  \f$q'\f$  has  to  be  replaced  by
    !! \f$-\iota'\f$ and \f$\theta\f$ by \f$\zeta\f$.
    !!
    !! The interpolated \c Ti_X are then used to calculate \f$U\f$:
    !!  \f[\begin{aligned}
    !!      U_0 =& -T_2                                  &\text{(order 1)}\\
    !!          &+ \frac{i}{n}\left(T_3 + i(nq-m) T_4\right) &\text{(order 2)}\\
    !!          &+ \left(\frac{i}{n}\right)^2 i(nq-m)
    !!          \left(-T_5 - (nq-m) T_6\right)           &\text{(order 3)}\\
    !!      U_1 =& \frac{i}{n}                           &\text{(order 1)}\\
    !!          &+ \left(\frac{i}{n}\right)^2 i(nq-m)(-T_1)  &\text{(order 2)} .
    !!  \end{aligned}\f]
    !!
    !! This is  valid for poloidal Flux  coordinates and where \f$n\f$  is to be
    !! replaced by  \f$m\f$ and \f$(nq-m)\f$  by \f$(n-\iota m)\f$  for toroidal
    !! Flux coordinates.
    !!
    !! For VMEC, these factors are also derived in the parallel coordinate.
    !!
    !! \see See \cite weyens2014theory.
    !!
    !! \return ierr
    integer function calc_u(grid_eq,grid_X,eq_1,eq_2,X) result(ierr)
        use num_vars, only: use_pol_flux_f, eq_style, u_style, &
            &norm_disc_prec_x, x_grid_style
        use num_utilities, only: c, spline
        use input_utilities, only: get_log, pause_prog
        use eq_vars, only: vac_perm
#if ldebug
        use num_vars, only: ltest
#endif

        character(*), parameter :: rout_name = 'calc_U'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                              !< equilibrium grid variables
        type(grid_type), intent(in) :: grid_x                               !< perturbation grid variables
        type(eq_1_type), intent(in), target :: eq_1                         !< flux equilibrium
        type(eq_2_type), intent(in), target :: eq_2                         !< metric equilibrium
        type(x_1_type), intent(inout) :: x                                  !< vectorial perturbation variables

        ! local variables
        integer :: id, jd, kd, ld                                           ! counters
        integer :: t_size                                                   ! 2 for VMEC and 1 for HELENA

        ! Jacobian
        real(dp), pointer :: j(:,:,:)                                       ! Jacobian
        real(dp), pointer :: d3j(:,:,:)                                     ! D_theta Jacobian
        ! lower metric factors
        real(dp), pointer :: g13(:,:,:)                                     ! g_alpha,theta
        real(dp), pointer :: d3g13(:,:,:)                                   ! D_theta g_alpha,theta
        real(dp), pointer :: g23(:,:,:)                                     ! g_psi,theta
        real(dp), pointer :: d3g23(:,:,:)                                   ! D_theta g_psi,theta
        real(dp), pointer :: g33(:,:,:)                                     ! g_theta,theta
        real(dp), pointer :: d3g33(:,:,:)                                   ! D_theta g_theta,theta
        ! upper metric factors
        real(dp), pointer :: h12(:,:,:)                                     ! h^alpha,psi
        real(dp), pointer :: d3h12(:,:,:)                                   ! D_theta h^alpha,psi
        real(dp), pointer :: h22(:,:,:)                                     ! h^psi,psi
        real(dp), pointer :: d1h22(:,:,:)                                   ! D_alpha h^psi,psi
        real(dp), pointer :: d3h22(:,:,:)                                   ! D_theta h^psi,psi
        real(dp), pointer :: d13h22(:,:,:)                                  ! D_alpha,theta h^psi,psi
        real(dp), pointer :: d33h22(:,:,:)                                  ! D_theta,theta h^psi,psi
        real(dp), pointer :: h23(:,:,:)                                     ! h^psi,theta
        real(dp), pointer :: d1h23(:,:,:)                                   ! D_alpha h^psi,theta
        real(dp), pointer :: d3h23(:,:,:)                                   ! D_theta h^psi,theta
        real(dp), pointer :: d13h23(:,:,:)                                  ! D_alpha,theta h^psi,theta
        real(dp), pointer :: d33h23(:,:,:)                                  ! D_theta,theta h^psi,theta
        ! helper variables
        real(dp), pointer :: ang_par_f(:,:,:)                               ! equilibrium parallel angle in flux coordinates
        real(dp), pointer :: ang_geo_f(:,:,:)                               ! equilibrium geodesical angle in flux coordinates
        real(dp), pointer :: q_saf(:), rot_t(:)                             ! safety factor and rotational transform in
```

```
X grid
        real(dp), allocatable :: djq(:)                                         ! either q' (pol. flux) or -iota' (tor. flux)
        real(dp), allocatable :: theta_3(:,:,:), d1theta_3(:,:,:), &
            &D3Theta_3(:,:,:)                                                   ! Theta^theta and derivatives
        real(dp), allocatable :: d13theta_3(:,:,:), d33theta_3(:,:,:)           ! Theta^theta and derivatives
        real(dp), allocatable :: n_frac(:,:)                                    ! nq-m (pol. flux) or n-iotam (tor. flux) for all modes
        real(dp), allocatable :: nm(:,:)                                        ! either n*A_0 (pol. flux) or m (tor.flux)
        complex(dp), allocatable :: u_corr(:,:,:,:)                             ! correction to U_0 and U_1 for a certain (n,m)
        complex(dp), allocatable :: d3u_corr(:,:,:,:)                           ! D_theta U_corr
        ! U factors
        real(dp), allocatable, target :: t1(:,:,:,:)                            ! B_alpha/B_theta and par. deriv.
        real(dp), allocatable, target :: t2(:,:,:,:)                            ! Theta^alpha + q' theta and par. deriv.
        real(dp), allocatable, target :: t3(:,:,:,:)                            ! B_alpha/B_theta q' + J^2/B_theta mu_0 p' and par. deriv.
        real(dp), allocatable, target :: t4(:,:,:,:)                            ! B_alpha/B_theta q' theta - B_psi/B_theta and par. deriv.
        real(dp), allocatable, target :: t5(:,:,:,:)                            ! B_alpha/B_theta D3Theta^theta - D1Theta^theta and par. deriv.
        real(dp), allocatable, target :: t6(:,:,:,:)                            ! B_alpha/B_theta Theta^theta and par. deriv.
        real(dp), pointer :: t1_x(:,:,:,:)                                      ! T1 in X grid and par. deriv.
        real(dp), pointer :: t2_x(:,:,:,:)                                      ! T2 in X grid and par. deriv.
        real(dp), pointer :: t3_x(:,:,:,:)                                      ! T3 in X grid and par. deriv.
        real(dp), pointer :: t4_x(:,:,:,:)                                      ! T4 in X grid and par. deriv.
        real(dp), pointer :: t5_x(:,:,:,:)                                      ! T5 in X grid and par. deriv.
        real(dp), pointer :: t6_x(:,:,:,:)                                      ! T6 in X grid and par. deriv.

        ! initialize ierr
        ierr = 0

        ! allocate variables
        ! helper variables
        allocate(nm(grid_x%loc_n_r,x%n_mod))
        allocate(q_saf(grid_x%loc_n_r))
        allocate(rot_t(grid_x%loc_n_r))
        allocate(djq(grid_eq%loc_n_r))
        allocate(theta_3(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
        allocate(d1theta_3(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
        allocate(d3theta_3(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
        allocate(d13theta_3(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
        allocate(d33theta_3(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
        allocate(n_frac(grid_x%loc_n_r,x%n_mod))
        allocate(u_corr(grid_x%n(1),grid_x%n(2),grid_x%loc_n_r,2))
        allocate(d3u_corr(grid_x%n(1),grid_x%n(2),grid_x%loc_n_r,2))
        ! U factors
        select case (eq_style)
            case (1)                                                            ! VMEC
                t_size = 2
            case (2)                                                            ! HELENA
                t_size = 1
        end select
        allocate(t1(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r,t_size))
        allocate(t2(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r,t_size))
        allocate(t3(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r,t_size))
        allocate(t4(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r,t_size))
        allocate(t5(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r,t_size))
        allocate(t6(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r,t_size))
        select case (x_grid_style)
            case (1)                                                            ! equilibrium
                ! do nothing
            case (2,3)                                                          ! solution or enriched
                allocate(q_saf(grid_x%loc_n_r))
                allocate(rot_t(grid_x%loc_n_r))
                allocate(t1_x(grid_x%n(1),grid_x%n(2),grid_x%loc_n_r,t_size))
                allocate(t2_x(grid_x%n(1),grid_x%n(2),grid_x%loc_n_r,t_size))
                allocate(t3_x(grid_x%n(1),grid_x%n(2),grid_x%loc_n_r,t_size))
                allocate(t4_x(grid_x%n(1),grid_x%n(2),grid_x%loc_n_r,t_size))
                allocate(t5_x(grid_x%n(1),grid_x%n(2),grid_x%loc_n_r,t_size))
                allocate(t6_x(grid_x%n(1),grid_x%n(2),grid_x%loc_n_r,t_size))
        end select

        ! set pointers
        if (use_pol_flux_f) then
            ang_par_f => grid_eq%theta_F
            ang_geo_f => grid_eq%zeta_F
        else
            ang_par_f => grid_eq%zeta_F
            ang_geo_f => grid_eq%theta_F
        end if
        j => eq_2%jac_FD(:,:,:,0,0,0)
        d3j => eq_2%jac_FD(:,:,:,0,0,1)
        g13 => eq_2%g_FD(:,:,:,c([1,3],.true.),0,0,0)
        d3g13 => eq_2%g_FD(:,:,:,c([1,3],.true.),0,0,1)
        g23 => eq_2%g_FD(:,:,:,c([2,3],.true.),0,0,0)
```

```fortran
        d3g23 => eq_2%g_FD(:,:,:,c([2,3],.true.),0,0,1)
        g33 => eq_2%g_FD(:,:,:,c([3,3],.true.),0,0,0)
        d3g33 => eq_2%g_FD(:,:,:,c([3,3],.true.),0,0,1)
        h12 => eq_2%h_FD(:,:,:,c([1,2],.true.),0,0,0)
        d3h12 => eq_2%h_FD(:,:,:,c([1,2],.true.),0,0,1)
        h22 => eq_2%h_FD(:,:,:,c([2,2],.true.),0,0,0)
        d1h22 => eq_2%h_FD(:,:,:,c([2,2],.true.),1,0,0)
        d3h22 => eq_2%h_FD(:,:,:,c([2,2],.true.),0,0,1)
        d13h22 => eq_2%h_FD(:,:,:,c([2,2],.true.),1,0,1)
        d33h22 => eq_2%h_FD(:,:,:,c([2,2],.true.),0,0,2)
        h23 => eq_2%h_FD(:,:,:,c([2,3],.true.),0,0,0)
        d1h23 => eq_2%h_FD(:,:,:,c([2,3],.true.),1,0,0)
        d3h23 => eq_2%h_FD(:,:,:,c([2,3],.true.),0,0,1)
        d13h23 => eq_2%h_FD(:,:,:,c([2,3],.true.),1,0,1)
        d33h23 => eq_2%h_FD(:,:,:,c([2,3],.true.),0,0,2)

        ! set up helper variables in eq grid
        if (use_pol_flux_f) then
            nm = x%n
            djq = eq_1%q_saf_FD(:,1)
        else
            djq = -eq_1%rot_t_FD(:,1)
            nm = x%m
        end if
        theta_3 = h23/h22
        select case (eq_style)
            case (1)                                                        ! VMEC
                d1theta_3 = d1h23/h22 - h23*d1h22/(h22**2)
                d3theta_3 = d3h23/h22 - h23*d3h22/(h22**2)
                d13theta_3 = d13h23/h22 - (d3h23*d1h22+d1h23*d3h22)/(h22**2) &
                    &- h23*d13h22/(h22**2) + 2*h23*d3h22*d1h22/(h22**3)
                d33theta_3 = d33h23/h22 - 2*d3h23*d3h22/(h22**2) &
                    &- h23*d33h22/(h22**2) + 2*h23*d3h22**2/(h22**3)
            case (2)                                                        ! HELENA
                ! geodesic derivative
                if (grid_eq%n(2).gt.norm_disc_prec_x+3) then                ! need enough terms
                    do kd = 1,grid_eq%loc_n_r
                        do id = 1,grid_eq%n(1)
                            ierr = spline(ang_geo_f(id,:,kd),theta_3(id,:,kd),&
                                &ang_geo_f(id,:,kd),d1theta_3(id,:,kd),&
                                &ord=norm_disc_prec_x,deriv=1)
                            chckerr('')
                        end do
                    end do
                else
                    d1theta_3 = 0._dp
                end if
                ! parallel derivative
                if (grid_eq%n(1).gt.norm_disc_prec_x+3) then                ! need enough terms
                    do kd = 1,grid_eq%loc_n_r
                        do jd = 1,grid_eq%n(2)
                            ierr = spline(ang_par_f(:,jd,kd),theta_3(:,jd,kd),&
                                &ang_par_f(:,jd,kd),d3theta_3(:,jd,kd),&
                                &ord=norm_disc_prec_x,deriv=1)
                            chckerr('')
                        end do
                    end do
                else
                    d3theta_3 = 0._dp
                end if
        end select

        ! set up U factors in eq grid
        t1(:,:,:,1) = g13/g33
        do kd = 1,grid_eq%loc_n_r
            t2(:,:,kd,1) = h12(:,:,kd)/h22(:,:,kd) + djq(kd)*ang_par_f(:,:,kd)
            t3(:,:,kd,1) = t1(:,:,kd,1)*djq(kd) + &
                &j(:,:,kd)**2*vac_perm*eq_1%pres_FD(kd,1)/g33(:,:,kd)
            t4(:,:,kd,1) = t1(:,:,kd,1)*djq(kd)*ang_par_f(:,:,kd) - &
                &g23(:,:,kd)/g33(:,:,kd)
        end do
        t5(:,:,:,1) = t1(:,:,:,1)*d3theta_3 - d1theta_3
        t6(:,:,:,1) = t1(:,:,:,1)*theta_3

        ! set up parallel derivatives of U in eq grid if VMEC is used
        if (eq_style.eq.1) then
            t1(:,:,:,2) = d3g13/g33 - g13*d3g33/g33**2
            do kd = 1,grid_eq%loc_n_r
                t2(:,:,kd,2) = d3h12(:,:,kd)/h22(:,:,kd) - &
                    &h12(:,:,kd)*d3h22(:,:,kd)/h22(:,:,kd)**2 + djq(kd)
                t3(:,:,kd,2) = t1(:,:,kd,2)*djq(kd) + &
                    &j(:,:,kd)*vac_perm*eq_1%pres_FD(kd,1)/g33(:,:,kd) * &
                    &(2*d3j(:,:,kd)-d3g33(:,:,kd)*j(:,:,kd)/g33(:,:,kd))
                t4(:,:,kd,2) = (t1(:,:,kd,2)*ang_par_f(:,:,kd)+t1(:,:,kd,1))*&
                    &djq(kd) - d3g23(:,:,kd)/g33(:,:,kd) + &
                    &g23(:,:,kd)*d3g33(:,:,kd)/g33(:,:,kd)**2
            end do
```

```
            t5(:,:,:,2) = t1(:,:,:,2)*d3theta_3 + t1(:,:,:,1)*d33theta_3 - &
                &d13theta_3
            t6(:,:,:,2) = t1(:,:,:,2)*theta_3 + t1(:,:,:,1)*d3theta_3
        end if

        select case (x_grid_style)
            case (1)                                                        ! equilibrium
                ! point
                q_saf => eq_1%q_saf_FD(:,0)
                rot_t => eq_1%rot_t_FD(:,0)
                t1_x => t1
                t2_x => t2
                t3_x => t3
                t4_x => t4
                t5_x => t5
                t6_x => t6
            case (2,3)                                                      ! solution or enriched
                ! interpolate
                ierr = spline(grid_eq%loc_r_F,eq_1%q_saf_FD(:,0),&
                    &grid_x%loc_r_F,q_saf,ord=norm_disc_prec_x)
                chckerr('')
                ierr = spline(grid_eq%loc_r_F,eq_1%rot_t_FD(:,0),&
                    &grid_x%loc_r_F,rot_t,ord=norm_disc_prec_x)
                chckerr('')
                do ld = 1,t_size
                    do jd = 1,grid_x%n(2)
                        do id = 1,grid_x%n(1)
                            ierr = spline(grid_eq%loc_r_F,t1(id,jd,:,ld),&
                                &grid_x%loc_r_F,t1_x(id,jd,:,ld),&
                                &ord=norm_disc_prec_x)
                            chckerr('')
                            ierr = spline(grid_eq%loc_r_F,t2(id,jd,:,ld),&
                                &grid_x%loc_r_F,t2_x(id,jd,:,ld),&
                                &ord=norm_disc_prec_x)
                            chckerr('')
                            ierr = spline(grid_eq%loc_r_F,t3(id,jd,:,ld),&
                                &grid_x%loc_r_F,t3_x(id,jd,:,ld),&
                                &ord=norm_disc_prec_x)
                            chckerr('')
                            ierr = spline(grid_eq%loc_r_F,t4(id,jd,:,ld),&
                                &grid_x%loc_r_F,t4_x(id,jd,:,ld),&
                                &ord=norm_disc_prec_x)
                            chckerr('')
                            ierr = spline(grid_eq%loc_r_F,t5(id,jd,:,ld),&
                                &grid_x%loc_r_F,t5_x(id,jd,:,ld),&
                                &ord=norm_disc_prec_x)
                            chckerr('')
                            ierr = spline(grid_eq%loc_r_F,t6(id,jd,:,ld),&
                                &grid_x%loc_r_F,t6_x(id,jd,:,ld),&
                                &ord=norm_disc_prec_x)
                            chckerr('')
                        end do
                    end do
                end do
        end select

        ! set up n_frac
        do kd = 1,grid_x%loc_n_r
            if (use_pol_flux_f) then
                n_frac(kd,:) = x%n(kd,:)*q_saf(kd) - x%m(kd,:)
            else
                n_frac(kd,:) = -x%m(kd,:)*rot_t(kd) + x%n(kd,:)
            end if
        end do

        ! calculate U_0 and U_1 for all modes
        do ld = 1,x%n_mod
            ! calculate order 1 of U_0 and U_1
            if (u_style.ge.1) then
                x%U_0(:,:,:,ld) = -t2_x(:,:,:,1)
                do kd = 1,grid_x%loc_n_r
                    x%U_1(:,:,kd,ld) = iu/nm(kd,ld)
                end do
            end if
            ! include order 2
            if (u_style.ge.2) then
                do kd = 1,grid_x%loc_n_r
                    u_corr(:,:,kd,1) = t3_x(:,:,kd,1) + &
                        &iu*n_frac(kd,ld)*t4_x(:,:,kd,1)
                    u_corr(:,:,kd,2) = n_frac(kd,ld)/nm(kd,ld)*t1_x(:,:,kd,1)
                    x%U_0(:,:,kd,ld) = x%U_0(:,:,kd,ld) + iu/nm(kd,ld)*&
                        &u_corr(:,:,kd,1)
                    x%U_1(:,:,kd,ld) = x%U_1(:,:,kd,ld) + iu/nm(kd,ld)*&
                        &u_corr(:,:,kd,2)
                end do
            end if
            ! include order 3
```

```
            if (u_style.ge.3) then
                do kd = 1,grid_x%loc_n_r
                    u_corr(:,:,kd,1) = iu*n_frac(kd,ld)*&
                        &(-t5_x(:,:,kd,1)-iu*n_frac(kd,ld)*t6_x(:,:,kd,1))
                    x%U_0(:,:,kd,ld) = x%U_0(:,:,kd,ld) + (iu/nm(kd,ld))**2*&
                        &u_corr(:,:,kd,1)
                end do
            end if
            if (u_style.ge.4) then
                call writo('The geodesic perturbation U is implemented up to &
                    &order 3',warning=.true.)
            end if
        end do

        ! calculate DU_0 and DU_1, starting with first part
        select case (eq_style)
            case (1)                                                        ! VMEC
                do ld = 1,x%n_mod
                    ! calculate order 1 of DU_0 and DU_1
                    if (u_style.ge.1) then
                        x%DU_0(:,:,:,ld) = -t2_x(:,:,:,2)
                        x%DU_1(:,:,:,ld) = 0._dp
                    end if
                    ! include order 2
                    if (u_style.ge.2) then
                        do kd = 1,grid_x%loc_n_r
                            u_corr(:,:,kd,1) = t3_x(:,:,kd,2) + &
                                &iu*n_frac(kd,ld)*t4_x(:,:,kd,2)
                            u_corr(:,:,kd,2) = n_frac(kd,ld)/nm(kd,ld)*&
                                &t1_x(:,:,kd,2)
                            x%DU_0(:,:,kd,ld) = x%DU_0(:,:,kd,ld) + &
                                &iu/nm(kd,ld)*u_corr(:,:,kd,1)
                            x%DU_1(:,:,kd,ld) = x%DU_1(:,:,kd,ld) + &
                                &iu/nm(kd,ld)*u_corr(:,:,kd,2)
                        end do
                    end if
                    ! include order 3
                    if (u_style.ge.3) then
                        do kd = 1,grid_x%loc_n_r
                            u_corr(:,:,kd,1) = iu*n_frac(kd,ld)*&
                                &(-t5_x(:,:,kd,2)-iu*n_frac(kd,ld)*&
                                &t6_x(:,:,kd,2))
                            x%DU_0(:,:,kd,ld) = x%DU_0(:,:,kd,ld) + &
                                &(iu/nm(kd,ld))**2*u_corr(:,:,kd,1)
                        end do
                    end if
                    if (u_style.ge.4) then
                        call writo('The geodesic perturbation U is implemented &
                            &only up to order 3',warning=.true.)
                    end if
                end do
#if ldebug
                if (ltest) then
                    call writo('Test calculation of DU')
                    if(get_log(.false.,ind=.true.)) then
                        ierr = test_du()
                        chckerr('')
                        call pause_prog(ind=.true.)
                    end if
                end if
#endif
            case (2)                                                        ! HELENA
                ! reset pointers
                if (use_pol_flux_f) then
                    ang_par_f => grid_x%theta_F
                else
                    ang_par_f => grid_x%zeta_F
                end if
                ! set up helper variable for derivative
                ! numerically derive U_0 and U_1
                do ld = 1,x%n_mod
                    do kd = 1,grid_x%loc_n_r
                        do jd = 1,grid_x%n(2)
                            ierr = spline(ang_par_f(:,jd,kd),x%U_0(:,jd,kd,ld),&
                                &ang_par_f(:,jd,kd),x%DU_0(:,jd,kd,ld),&
                                &ord=norm_disc_prec_x,deriv=1)
                            chckerr('')
                            ierr = spline(ang_par_f(:,jd,kd),x%U_1(:,jd,kd,ld),&
                                &ang_par_f(:,jd,kd),x%DU_1(:,jd,kd,ld),&
                                &ord=norm_disc_prec_x,deriv=1)
                            chckerr('')
                        end do
                    end do
                end do
        end select

        ! add second part of derivatives ~ n_frac
```

```
        do kd = 1,grid_x%loc_n_r
            do ld = 1,x%n_mod
                x%DU_0(:,:,kd,ld) = x%DU_0(:,:,kd,ld) + &
                    &iu*n_frac(kd,ld)*x%U_0(:,:,kd,ld)
                x%DU_1(:,:,kd,ld) = x%DU_1(:,:,kd,ld) + &
                    &iu*n_frac(kd,ld)*x%U_1(:,:,kd,ld)
            end do
        end do

        ! clean up
        nullify(ang_par_f,ang_geo_f)
        nullify(j,d3j)
        nullify(g13,d3g13)
        nullify(g23,d3g23)
        nullify(g33,d3g33)
        nullify(h12,d3h12)
        nullify(h22,d1h22,d3h22,d13h22,d33h22)
        nullify(h23,d1h23,d3h23,d13h23,d33h23)
        select case (x_grid_style)
            case (1)                                                    ! equilibrium
                ! do nothing
            case (2,3)                                                  ! solution or enriched
                deallocate(q_saf,rot_t,t1_x,t2_x,t3_x,t4_x,t5_x,t6_x)
        end select
        nullify(q_saf,rot_t,t1_x,t2_x,t3_x,t4_x,t5_x,t6_x)
#if ldebug
    contains
        ! Test calculation of DU by deriving U numerically.
        integer function test_du() result(ierr)
            use num_vars, only: norm_disc_prec_x

            character(*), parameter :: rout_name = 'test_DU'

            ! local variables
            integer :: jd, kd, ld                                       ! counters
            complex(dp), allocatable :: du_0(:,:,:)                     ! alternative calculation for DU_0
            complex(dp), allocatable :: du_1(:,:,:)                     ! alternative calculation for DU_1
            character(len=max_str_ln) :: file_name                      ! name of plot file
            character(len=max_str_ln) :: description                    ! description of plot

            ! initialize ierr
            ierr = 0

            ! warning
            call writo('This test is representable only if there are enough &
                &parallel points in the grid!',warning=.true.)

            ! output
            call writo('Going to test whether DU is consistent with U')
            call lvl_ud(1)

            ! set up DU_0 and DU_1
            allocate(du_0(grid_x%n(1),grid_x%n(2),grid_x%loc_n_r))
            allocate(du_1(grid_x%n(1),grid_x%n(2),grid_x%loc_n_r))

            ! reset pointers
            if (use_pol_flux_f) then
                ang_par_f => grid_x%theta_F
            else
                ang_par_f => grid_x%zeta_F
            end if

            ! loop over all modes
            do ld = 1,x%n_mod
                ! derivate numerically
                do kd = 1,grid_x%loc_n_r
                    do jd = 1,grid_x%n(2)
                        ierr = spline(ang_par_f(:,jd,kd),x%U_0(:,jd,kd,ld),&
                            &ang_par_f(:,jd,kd),du_0(:,jd,kd),&
                            &ord=norm_disc_prec_x,deriv=1)
                        chkerr('')
                        ierr = spline(ang_par_f(:,jd,kd),x%U_1(:,jd,kd,ld),&
                            &ang_par_f(:,jd,kd),du_1(:,jd,kd),&
                            &ord=norm_disc_prec_x,deriv=1)
                        chkerr('')
                    end do
                end do

                ! set some variables
                file_name = 'TEST_RE_DU_0_'//trim(i2str(ld))
                description = 'Verifying DU_0 by deriving U_0 numerically for &
                    &mode '//trim(i2str(ld))//', real part')

                ! plot difference for RE DU_0
                call plot_diff_hdf5(rp(du_0),rp(x%DU_0(:,:,:,ld)),&
                    &file_name,descr=description,output_message=.true.)

```

```
                ! set some variables
                file_name = 'TEST_IM_DU_0_'//trim(i2str(ld))
                description = 'Verifying DU_0 by deriving U_0 numerically for &
                    &mode '//trim(i2str(ld))//', imaginary part')

                ! plot difference for IM DU_0
                call plot_diff_hdf5(ip(du_0),ip(x%DU_0(:,:,:,ld)),&
                    &file_name,descr=description,output_message=.true.)

                ! set some variables
                file_name = 'TEST_RE_DU_1_'//trim(i2str(ld))
                description = 'Verifying DU_1 by deriving U_1 numerically for &
                    &mode '//trim(i2str(ld))//', real part')

                ! plot difference for RE DU_1
                call plot_diff_hdf5(rp(du_1),rp(x%DU_1(:,:,:,ld)),&
                    &file_name,descr=description,output_message=.true.)

                ! set some variables
                file_name = 'TEST_IM_DU_1_'//trim(i2str(ld))
                description = 'Verifying DU_1 by deriving U_1 numerically for &
                    &mode '//trim(i2str(ld))//', imaginary part')

                ! plot difference for IM DU_1
                call plot_diff_hdf5(ip(du_1),ip(x%DU_1(:,:,:,ld)),&
                    &file_name,descr=description,output_message=.true.)
            end do

            ! user output
            call lvl_ud(-1)
            call writo('Test complete')
        end function test_du
#endif
    end function calc_u

    !> calculate      \f$\widetilde{PV}_{k,m}^i\f$      (pol.      flux)      or
    !! \f$\widetilde{PV}_{l,n}^i\f$ (tor.  flux) at  all equilibrium  \c loc_n_r
    !! values.
    !!
    !! Like in calc_U(), use  is made of variables \c Ti that are  set up in the
    !! equilibrium  grid  and  are  afterwards  converted  to  \c  Ti_X  in  the
    !! perturbation grid,  which needs to  have the same angular  coordinates as
    !! the equilibrium grid:
    !!  \f[\begin{aligned}
    !!      T_1 &= \frac{h^{22}}{\mu_0} g_{33} \\
    !!      T_2 &= \mathcal{J}S +
    !!          \mu_0 \sigma \frac{g_{33}}{\mathcal{J}} h^{22} \\
    !!      T_3 &= \frac{\sigma}{\mathcal{J}} T_2 \\
    !!      T_4 &= \frac{1}{\mu_0} \mathcal{J}^2 h^{22} \\
    !!      T_5 &= 2 p' \kappa_n \ ,
    !!  \end{aligned}\f]
    !! with \f$T_i\f$ = \c Ti.
    !!
    !! The     interpolated    Ti_X     are    then     used    to     calculate
    !! \f$\widetilde{PV}_{k,m}^i\f$:
    !!  \f[\begin{aligned}
    !!      \widetilde{PV}_{k,m}^0 &= T_1(DU_k^{0*} - T_2)(DU_m^0 - T_2) -
    !!          T_3 + (nq-m)(nq-k)T_4 - T_5 \\
    !!      \widetilde{PV}_{k,m}^1 &= T_1(DU_k^{0*} - T_2) DU_m^1 \\
    !!      \widetilde{PV}_{k,m}^2 &= T_1 DU_k^{1*} DU_m^1 \ ,
    !!  \end{aligned}\f]
    !! where
    !!  \f[DU_k^i = i (nq-m) U_k^i + \frac{\partial U_k^i}{\partial\theta} . \f]
    !!
    !! This is  valid for poloidal Flux  coordinates and where \f$n\f$  is to be
    !! replaced by  \f$m\f$ and \f$(nq-m)\f$  by \f$(n-\iota m)\f$  for toroidal
    !! Flux coordinates.
    !!
    !! \see See \cite weyens2014theory .
    !!
    !! \return ierr
    integer function calc_pv(grid_eq,grid_X,eq_1,eq_2,X_a,X_b,X,lim_sec_X) &
        &result(ierr)
        use num_vars, only: use_pol_flux_f, norm_disc_prec_x, x_grid_style
        use eq_vars, only: vac_perm
        use num_utilities, only: c, spline
        use x_utilities, only: is_necessary_x
        use x_vars, only: n_mod_x

        character(*), parameter :: rout_name = 'calc_PV'

        ! use input / output
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(grid_type), intent(in) :: grid_x                                   !< perturbation grid
        type(eq_1_type), intent(in), target :: eq_1                             !< flux equilibrium
        type(eq_2_type), intent(in), target :: eq_2                             !< metric equilibrium
        type(x_1_type), intent(in) :: x_a                                       !< vectorial perturbation variables of
```

```
dimension 2
        type(x_1_type), intent(in) :: x_b                                   !< vectorial perturbation variables of dimension 2
        type(x_2_type), intent(inout) :: x                                  !< tensorial perturbation variables
        integer, intent(in), optional :: lim_sec_x(2,2)                     !< limits of \c m_X (pol flux) or \c n_X (tor flux) for both dimensions

        ! local variables
        integer :: m, k                                                     ! counters
        integer :: id, jd, kd                                               ! counters
        integer :: c_loc(2)                                                 ! local c for symmetric and asymmetric variables
        logical :: calc_this(2)                                             ! whether this combination needs to be calculated

        ! jacobian
        real(dp), pointer :: j(:,:,:)                                       ! jac
        ! lower metric factors
        real(dp), pointer :: g33(:,:,:)                                     ! h_theta,theta or h_zeta,zeta
        ! upper metric factors
        real(dp), pointer :: h22(:,:,:)                                     ! h^psi,psi
        ! helper variables
        real(dp), pointer :: q_saf(:), rot_t(:)                             ! safety factor and rotational transform in X grid
        real(dp), allocatable :: fac_n(:), fac_m(:)                         ! multiplicative factors for n and m
        ! PV factors
        real(dp), allocatable, target :: t1(:,:,:)                          ! h22/mu_0 g33
        real(dp), allocatable, target :: t2(:,:,:)                          ! JS + mu_0 sigma g33/J h22
        real(dp), allocatable, target :: t3(:,:,:)                          ! sigma/J T2
        real(dp), allocatable, target :: t4(:,:,:)                          ! 1/mu_0 J^2 h22
        real(dp), allocatable, target :: t5(:,:,:)                          ! 2 p' kappa_n
        real(dp), pointer :: t1_x(:,:,:)                                    ! T1 in X grid and par. deriv.
        real(dp), pointer :: t2_x(:,:,:)                                    ! T2 in X grid and par. deriv.
        real(dp), pointer :: t3_x(:,:,:)                                    ! T3 in X grid and par. deriv.
        real(dp), pointer :: t4_x(:,:,:)                                    ! T4 in X grid and par. deriv.
        real(dp), pointer :: t5_x(:,:,:)                                    ! T5 in X grid and par. deriv.

        ! initialize ierr
        ierr = 0

        ! allocate variables
        ! helper variables
        allocate(q_saf(grid_x%loc_n_r))
        allocate(rot_t(grid_x%loc_n_r))
        allocate(fac_n(grid_x%loc_n_r),fac_m(grid_x%loc_n_r))
        ! PV factors
        allocate(t1(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
        allocate(t2(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
        allocate(t3(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
        allocate(t4(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
        allocate(t5(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
        select case (x_grid_style)
            case (1)                                                        ! equilibrium
                ! do nothing
            case (2,3)                                                      ! solution or enriched
                allocate(q_saf(grid_x%loc_n_r))
                allocate(rot_t(grid_x%loc_n_r))
                allocate(t1_x(grid_x%n(1),grid_x%n(2),grid_x%loc_n_r))
                allocate(t2_x(grid_x%n(1),grid_x%n(2),grid_x%loc_n_r))
                allocate(t3_x(grid_x%n(1),grid_x%n(2),grid_x%loc_n_r))
                allocate(t4_x(grid_x%n(1),grid_x%n(2),grid_x%loc_n_r))
                allocate(t5_x(grid_x%n(1),grid_x%n(2),grid_x%loc_n_r))
        end select

        ! set pointers
        j => eq_2%jac_FD(:,:,:,0,0,0)
        g33 => eq_2%g_FD(:,:,:,c([3,3],.true.),0,0,0)
        h22 => eq_2%h_FD(:,:,:,c([2,2],.true.),0,0,0)

        ! set up PV factors in eq grid
        t1 = h22/(vac_perm*g33)
        t2 = j*eq_2%S + vac_perm*eq_2%sigma*g33/(j*h22)
        t3 = eq_2%sigma/j*t2
        t4 = 1._dp/(vac_perm*j**2*h22)
        do kd = 1,grid_eq%loc_n_r
            t5(:,:,kd) = 2*eq_1%pres_FD(kd,1)*eq_2%kappa_n(:,:,kd)
        end do

        select case (x_grid_style)
            case (1)                                                        ! equilibrium
                ! point
                q_saf => eq_1%q_saf_FD(:,0)
                rot_t => eq_1%rot_t_FD(:,0)
                t1_x => t1
                t2_x => t2
                t3_x => t3
                t4_x => t4
```

```
                t5_x => t5
            case (2,3)                                                          ! solution or enriched
                ! interpolate
                ierr = spline(grid_eq%loc_r_F,eq_1%q_saf_FD(:,0),&
                    &grid_x%loc_r_F,q_saf,ord=norm_disc_prec_x)
                chckerr('')
                ierr = spline(grid_eq%loc_r_F,eq_1%rot_t_FD(:,0),&
                    &grid_x%loc_r_F,rot_t,ord=norm_disc_prec_x)
                chckerr('')
                do jd = 1,grid_x%n(2)
                    do id = 1,grid_x%n(1)
                        ierr = spline(grid_eq%loc_r_F,t1(id,jd,:),&
                            &grid_x%loc_r_F,t1_x(id,jd,:),&
                            &ord=norm_disc_prec_x)
                        chckerr('')
                        ierr = spline(grid_eq%loc_r_F,t2(id,jd,:),&
                            &grid_x%loc_r_F,t2_x(id,jd,:),&
                            &ord=norm_disc_prec_x)
                        chckerr('')
                        ierr = spline(grid_eq%loc_r_F,t3(id,jd,:),&
                            &grid_x%loc_r_F,t3_x(id,jd,:),&
                            &ord=norm_disc_prec_x)
                        chckerr('')
                        ierr = spline(grid_eq%loc_r_F,t4(id,jd,:),&
                            &grid_x%loc_r_F,t4_x(id,jd,:),&
                            &ord=norm_disc_prec_x)
                        chckerr('')
                        ierr = spline(grid_eq%loc_r_F,t5(id,jd,:),&
                            &grid_x%loc_r_F,t5_x(id,jd,:),&
                            &ord=norm_disc_prec_x)
                        chckerr('')
                    end do
                end do
        end select

        ! set up fac_n and fac_m
        if (use_pol_flux_f) then
            fac_n = q_saf
            fac_m = 1.0_dp
        else
            fac_n = 1.0_dp
            fac_m = rot_t
        end if

        ! loop over all modes
        do m = 1,x_b%n_mod
            do k = 1,x_a%n_mod
                ! check whether modes are correct
                do kd = 1,grid_x%loc_n_r
                    if (x%n_1(kd,k).ne.x_a%n(kd,k) .or. &
                        &x%n_2(kd,m).ne.x_b%n(kd,m) .or. &
                        &x%m_1(kd,k).ne.x_a%m(kd,k) .or. &
                        &x%m_2(kd,m).ne.x_b%m(kd,m)) then
                        ierr = 1
                        chckerr('Modes do not match')
                    end if
                end do

                ! check whether mode combination needs to be calculated
                calc_this(1) = is_necessary_x(.true.,[k,m],lim_sec_x)
                calc_this(2) = is_necessary_x(.false.,[k,m],lim_sec_x)

                ! set up c_loc
                c_loc(1) = c([k,m],.true.,n_mod_x,lim_sec_x)
                c_loc(2) = c([k,m],.false.,n_mod_x,lim_sec_x)

                ! calculate PV_0
                if (calc_this(1)) then
                    do kd = 1,grid_x%loc_n_r
                        x%PV_0(:,:,kd,c_loc(1)) = t1_x(:,:,kd)*&
                            &(x_b%DU_0(:,:,kd,m) - t2_x(:,:,kd)) * &
                            &(conjg(x_a%DU_0(:,:,kd,k)) - t2_x(:,:,kd)) - &
                            &t3_x(:,:,kd) - t5_x(:,:,kd) + t4_x(:,:,kd)*&
                            &(x_b%n(kd,m)*fac_n(kd)-x_b%m(kd,m)*fac_m(kd))*&
                            &(x_a%n(kd,k)*fac_n(kd)-x_a%m(kd,k)*fac_m(kd))
                    end do
                end if

                ! calculate PV_1
                if (calc_this(2)) then
                    x%PV_1(:,:,:,c_loc(2)) = t1_x * x_b%DU_1(:,:,:,m) * &
                        &(conjg(x_a%DU_0(:,:,:,k))-t2_x)
                end if

                ! calculate PV_2
                if (calc_this(1)) then
                    x%PV_2(:,:,:,c_loc(1)) = t1_x * x_b%DU_1(:,:,:,m) * &
```

```
                        &conjg(x_a%DU_1(:,:,:,k))
                end if
            end do
        end do

        ! clean up
        nullify(j)
        nullify(g33)
        nullify(h22)
        select case (x_grid_style)
            case (1)                                                    ! equilibrium
                ! do nothing
            case (2,3)                                                  ! solution or enriched
                deallocate(q_saf,rot_t,t1_x,t2_x,t3_x,t4_x,t5_x)
        end select
        nullify(q_saf,rot_t,t1_x,t2_x,t3_x,t4_x,t5_x)
    end function calc_pv

    !> calculate      \f$\widetilde{KV}_{k,m}^i\f$      (pol.      flux)      or
    !! \f$\widetilde{KV}_{l,n}^i\f$ (tor.  flux) at  all equilibrium  \c loc_n_r
    !! values.
    !!
    !! Like in calc_U(), use  is made of variables \c Ti that are  set up in the
    !! equilibrium  grid  and  are  afterwards  converted  to  \c  Ti_X  in  the
    !! perturbation grid,  which needs to  have the same angular  coordinates as
    !! the equilibrium grid:
    !!  \f[\begin{aligned}
    !!      T_1 &= \rho \mathcal{J}^2 \frac{h^{22}}{\mu_0} g_{33} \\
    !!      T_2 &= \rho \frac{1}{h^{22}} \ ,
    !!  \end{aligned}\f]
    !! with \f$T_i\f$ = \c Ti.
    !!
    !! The     interpolated    Ti_X     are     then     used     to     calculate
    !! \f$\widetilde{KV}_{k,m}^i\f$:
    !!  \f[\begin{aligned}
    !!      \widetilde{KV}_{k,m}^0 &= T_1 U_k^{0*} U_m^0 + T_2 \\
    !!      \widetilde{KV}_{k,m}^1 &= T_1 U_k^{0*} U_m^1 \\
    !!      \widetilde{KV}_{k,m}^2 &= T_1 U_k^{1*} U_m^1 \ ,
    !!  \end{aligned}\f]
    !! where
    !!  \f[DU_k^i = i (nq-m) U_k^i + \frac{\partial U_k^i}{\partial\theta} . \f]
    !!
    !! This is  valid for poloidal Flux  coordinates and where \f$n\f$  is to be
    !! replaced by  \f$m\f$ and \f$(nq-m)\f$  by \f$(n-\iota m)\f$  for toroidal
    !! Flux coordinates.
    !!
    !! \see See \cite weyens2014theory .
    !!
    !! \return ierr
    integer function calc_kv(grid_eq,grid_X,eq_1,eq_2,X_a,X_b,X,lim_sec_X) &
        &result(ierr)
        use num_vars, only: k_style, norm_disc_prec_x, x_grid_style
        use num_utilities, only: c, spline
        use x_utilities, only: is_necessary_x
        use x_vars, only: n_mod_x

        character(*), parameter :: rout_name = 'calc_KV'

        ! use input / output
        type(grid_type), intent(in) :: grid_eq                          !< equilibrium grid
        type(grid_type), intent(in) :: grid_x                           !< perturbation grid
        type(eq_1_type), intent(in) :: eq_1                             !< flux equilibrium
        type(eq_2_type), intent(in), target :: eq_2                     !< metric equilibrium
        type(x_1_type), intent(in) :: x_a                               !< vectorial perturbation variables of dimension 2
        type(x_1_type), intent(in) :: x_b                               !< vectorial perturbation variables of dimension 2
        type(x_2_type), intent(inout) :: x                              !< tensorial perturbation variables
        integer, intent(in), optional :: lim_sec_x(2,2)                 !< limits of \c m_X (pol flux) or \c n_X (tor flux) for both dimensions

        ! local variables
        integer :: m, k                                                 ! counters
        integer :: id, jd, kd                                           ! counters
        integer :: c_loc(2)                                             ! local c for symmetric and asymmetric variables
        logical :: calc_this(2)                                         ! whether this combination needs to be calculated

        ! jacobian
        real(dp), pointer :: j(:,:,:)                                   ! jac
        ! lower metric factors
        real(dp), pointer :: g33(:,:,:)                                 ! h_theta,theta or h_zeta,zeta
        ! upper metric factors
        real(dp), pointer :: h22(:,:,:)                                 ! h^psi,psi
        ! KV factors
        real(dp), allocatable, target :: t1(:,:,:)                      ! h22/mu_0 J g33
```

```
        real(dp), allocatable, target :: t2(:,:,:)                          ! JS + mu_0 sigma g33/h22
        real(dp), pointer :: t1_x(:,:,:)                                    ! T1 in X grid and par. deriv.
        real(dp), pointer :: t2_x(:,:,:)                                    ! T2 in X grid and par. deriv.

        ! initialize ierr
        ierr = 0

        ! allocate variables
        ! KV factors
        allocate(t1(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
        allocate(t2(grid_eq%n(1),grid_eq%n(2),grid_eq%loc_n_r))
        select case (x_grid_style)
            case (1)                                                        ! equilibrium
                ! do nothing
            case (2,3)                                                      ! solution or enriched
                allocate(t1_x(grid_x%n(1),grid_x%n(2),grid_x%loc_n_r))
                allocate(t2_x(grid_x%n(1),grid_x%n(2),grid_x%loc_n_r))
        end select

        ! set pointers
        j => eq_2%jac_FD(:,:,:,0,0,0)
        g33 => eq_2%g_FD(:,:,:,c([3,3],.true.),0,0,0)
        h22 => eq_2%h_FD(:,:,:,c([2,2],.true.),0,0,0)

        ! set up KV factors in eq grid
        do kd = 1,grid_eq%loc_n_r
            t1(:,:,kd) = eq_1%rho(kd)*j(:,:,kd)**2*h22(:,:,kd)/g33(:,:,kd)
            t2(:,:,kd) = eq_1%rho(kd)/h22(:,:,kd)
        end do

        select case (x_grid_style)
            case (1)                                                        ! equilibrium
                ! point
                t1_x => t1
                t2_x => t2
            case (2,3)                                                      ! solution or enriched
                ! interpolate
                do jd = 1,grid_x%n(2)
                    do id = 1,grid_x%n(1)
                        ierr = spline(grid_eq%loc_r_F,t1(id,jd,:),&
                            &grid_x%loc_r_F,t1_x(id,jd,:),&
                            &ord=norm_disc_prec_x)
                        chckerr('')
                        ierr = spline(grid_eq%loc_r_F,t2(id,jd,:),&
                            &grid_x%loc_r_F,t2_x(id,jd,:),&
                            &ord=norm_disc_prec_x)
                        chckerr('')
                    end do
                end do
        end select

        ! loop over all modes
        do m = 1,x_b%n_mod
            do k = 1,x_a%n_mod
                ! check whether modes are correct
                do kd = 1,grid_x%loc_n_r
                    if (x%n_1(kd,k).ne.x_a%n(kd,k) .or. &
                        &x%n_2(kd,m).ne.x_b%n(kd,m) .or. &
                        &x%m_1(kd,k).ne.x_a%m(kd,k) .or. &
                        &x%m_2(kd,m).ne.x_b%m(kd,m)) then
                        ierr = 1
                        chckerr('Modes do not match')
                    end if
                end do

                ! check whether mode combination needs to be calculated
                calc_this(1) = is_necessary_x(.true.,[k,m],lim_sec_x)
                calc_this(2) = is_necessary_x(.false.,[k,m],lim_sec_x)

                ! set up c_loc
                c_loc(1) = c([k,m],.true.,n_mod_x,lim_sec_x)
                c_loc(2) = c([k,m],.false.,n_mod_x,lim_sec_x)

                select case (k_style)
                    case (1)                                                ! normalization of full perpendicular
    component
                        ! calculate KV_0
                        if (calc_this(1)) then
                            x%KV_0(:,:,:,c_loc(1)) = t2_x + t1_x * &
                                &x_b%U_0(:,:,:,m) * conjg(x_a%U_0(:,:,:,k))
                        end if

                        ! calculate KV_1
                        if (calc_this(2)) then
                            x%KV_1(:,:,:,c_loc(2)) = t1_x * &
                                &x_b%U_1(:,:,:,m) * conjg(x_a%U_0(:,:,:,k))
                        end if
```

```

                        ! calculate KV_2
                        if (calc_this(1)) then
                            x%KV_2(:,:,:,c_loc(1)) = t1_x * &
                                &x_b%U_1(:,:,:,m) * conjg(x_a%U_1(:,:,:,k))
                        end if
                    case (2)                                                ! normalization of only normal component
                        ! calculate KV_0
                        if (calc_this(1)) then
                            x%KV_0(:,:,:,c_loc(1)) = t2_x
                        end if

                        ! calculate KV_1
                        if (calc_this(2)) then
                            x%KV_1(:,:,:,c_loc(2)) = 0._dp
                        end if

                        ! calculate KV_2
                        if (calc_this(1)) then
                            x%KV_2(:,:,:,c_loc(1)) = 0._dp
                        end if
                end select
            end do
        end do

        ! clean up
        nullify(j)
        nullify(g33)
        nullify(h22)
        select case (x_grid_style)
            case (1)                                                        ! equilibrium
                ! do nothing
            case (2,3)                                                      ! solution or enriched
                deallocate(t1_x,t2_x)
        end select
        nullify(t1_x,t2_x)
    end function calc_kv

    !> Calculate  the magnetic  integrals from  \f$\widetilde{PV}_{k,m}^i\f$ and
    !! \f$\widetilde{KV}_{k,m}^i\f$ in  an equidistant grid where  the step size
    !! can vary depending on the normal coordinate.
    !!
    !! All the variables should thus be field-line oriented. The result is saved
    !! in the first index of the X variables, the other can be ignored.
    !!
    !! Using \c prev_style, the results of this calculation on X can be combined
    !! with the ones already present in \c X_int.
    !!  - \c prev_style=0: Overwrite [def].
    !!  - \c prev_style=1: Add to current integral.
    !!  - \c prev_style=2: Divide by  2 and add to current integral.  Also modify the
    !!  indices of the current integral:
    !!      - for \c magn_int_style = 1: <tt>1 2 2 2 2 2 1 -> 2 2 2 2 2 2</tt>,
    !!      - for \c magn_int_style = 2: <tt>1 3 3 2 3 3 1 -> 3 2 3 3 2 3</tt>.
    !!  - \c prev_style=3:  Add to current integral. Also modify  the indices of
    !!  the current integral as in \c prev_style = 2.
    !!  - else: ignore previous magnetic integrals.
    !!
    !! Therefore, it is to be used in order. For example:
    !!  -# \f$\phantom{\rightarrow}\f$  (R=1,E=1): style 0,
    !!  -# \f$\rightarrow\f$            (R=1,E=2): style 1,
    !!  -# \f$\rightarrow\f$            (R=2,E=1): style 2,
    !!  -# \f$\rightarrow\f$            (R=2,E=2): style 3.
    !!
    !! Afterwards, it would cycle through 2 and 3,  as style 0 and 1 are only for
    !! the first Richardson level.
    !!
    !! \note
    !!  -# The variable type for the integrated tensorial perturbation variables
    !!  is also \c X_2, but it is  assumed that the first index has dimension 1,
    !!  the rest is ignored.
    !!  -# Simpson's 3/8  rule converges faster than the  trapezoidal rule, but
    !!  normally needs a better starting point (i.e. higher \c min_n_par_X)
    !!  -#  For alpha  style 1,  Weyl's  lemma is  used to  convert the  surface
    !!  integral  into a  line integral  along  the magnetic  field. This  means
    !!  that  the  resulting line  integral  still  needs  to be  multiplied  by
    !!  \f$\frac{2\pi}{M}\f$, where \f$M\f$ is the  number of times the poloidal
    !!  or  toroidal angle  completes a  full \f$2\pi\f$  tour, depending  on \c
    !!  use_pol_flux_F \cite Helander2014.
    !!  -# For alpha style 2, this factor is just the step size in alpha.
    !!
    !! \see See x_vars.x_2_type.
    !!
    !! \return ierr
    integer function calc_magn_ints(grid_eq,grid_X,eq,X,X_int,prev_style,&
        &lim_sec_X) result(ierr)
        use num_vars, only: use_pol_flux_f, norm_disc_prec_x, magn_int_style, &
            &alpha_style, x_grid_style
```

```
        use x_utilities, only: is_necessary_x
        use x_vars, only: n_mod_x
        use num_utilities, only: c, spline
        use grid_vars, only: min_par_x, max_par_x, n_alpha
        use rich_vars, only: rich_lvl

        character(*), parameter :: rout_name = 'calc_magn_ints'

        ! input / output
        type(grid_type), intent(in) :: grid_eq                                  !< equilibrium grid
        type(grid_type), intent(in) :: grid_x                                   !< perturbation grid
        type(eq_2_type), intent(in), target :: eq                               !< metric equilibrium
        type(x_2_type), intent(in) :: x                                         !< tensorial perturbation variables
        type(x_2_type), intent(inout) :: x_int                                  !< interpolated tensorial perturbation variables, containing all mode combinations
        integer, intent(in), optional :: prev_style                             !< style to treat X_prev
        integer, intent(in), optional :: lim_sec_x(2,2)                         !< limits of \c m_X (pol flux) or \c n_X (tor flux) for both dimensions

        ! local variables, not used in subroutines
        logical :: calc_this(2)                                                 ! whether this combination needs to be calculated
        integer :: k, m                                                         ! counters
        integer :: id, jd, kd                                                   ! counters
        integer :: c_loc(2)                                                     ! local c for symmetric and asymmetric variables
        integer :: c_tot(2)                                                     ! total c for symmetric and asymmetric variables
        integer :: prev_style_loc                                               ! local prev_style
        real(dp) :: alpha_fac                                                   ! factor for alpha (style 1: Weyl factor, style 2: step size)
        real(dp) :: prev_mult_fac                                               ! multiplicative factor for previous style
        real(dp), allocatable :: step_size(:,:)                                 ! step size for every field line and normal point
        real(dp), pointer :: ang_par_f(:,:,:) => null()                         ! parallel angle in flux coordinates
        complex(dp), allocatable :: v_int_work(:,:)                             ! work V_int

        ! local variables, also used in subroutines
        integer :: nr_int_regions                                               ! nr. of integration regions
        integer, allocatable :: int_dims(:,:)                                   ! dimension information of integration regions
        real(dp), allocatable :: int_facs(:)                                    ! integration factors for each of the regions
        complex(dp), allocatable :: j_exp_ang(:,:,:)                            ! J * exponential of Flux parallel angle

        ! Makes use of nr_int_regions, int_dims, int_facs and J_exp_ang

        ! jacobian
        real(dp), pointer :: j(:,:,:)                                           ! jac

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Calculate field-line averages')
        call lvl_ud(1)

        ! set up local prev_style
        prev_style_loc = 0
        if (present(prev_style)) prev_style_loc = prev_style

        ! allocate variables
        allocate(j_exp_ang(grid_x%n(1),grid_x%n(2),grid_x%loc_n_r))
        select case (x_grid_style)
            case (1)                                                            ! equilibrium
                ! point
                j => eq%jac_FD(:,:,:,0,0,0)
            case (2,3)                                                          ! solution or enriched
                ! allocate
                allocate(j(grid_eq%n(1),grid_eq%n(2),grid_x%loc_n_r))

                ! interpolate
                do jd = 1,grid_x%n(2)
                    do id = 1,grid_x%n(1)
                        ierr = spline(grid_eq%loc_r_F,eq%jac_FD(id,jd,:,0,0,0),&
                            &grid_x%loc_r_F,j(id,jd,:),ord=norm_disc_prec_x)
                        chckerr('')
                    end do
                end do
        end select

        ! set up parallel angle in flux coordinates
        if (use_pol_flux_f) then
            ang_par_f => grid_x%theta_F
        else
            ang_par_f => grid_x%zeta_F
        end if
```

```

        ! set up alpha factor
        select case (alpha_style)
            case (1)                                                    ! single field line, multiple turns
                alpha_fac = (2*pi)**2/((max_par_x-min_par_x)*pi)        ! Weyl factor
            case (2)                                                    ! multiple field lines, single turns
                alpha_fac = (2*pi)/n_alpha                              ! grid size
        end select

        ! set up integration variables

        ! set nr_int_regions
        select case (magn_int_style)
            case (1)                                                    ! Trapezoidal rule
                select case (prev_style_loc)
                    case (2,3)                                          ! change indices
                        nr_int_regions = 1
                    case default                                        ! don't change indices
                        nr_int_regions = 2
                end select
            case (2)                                                    ! Simpson's 3/8 rule
                select case (prev_style_loc)
                    case (2,3)                                          ! change indices
                        nr_int_regions = 3
                    case default                                        ! don't change indices
                        nr_int_regions = 4
                end select
        end select

        ! set up integration factors and dimensions
        allocate(int_facs(nr_int_regions))
        allocate(int_dims(nr_int_regions,3))
        select case (magn_int_style)
            case (1)                                                    ! Trapezoidal rule
                select case (prev_style_loc)
                    case (2,3)                                          ! change indices
                        int_facs = 0.5_dp*[2]
                        int_dims(1,:) = [1,grid_x%n(1),1]               ! region 1: all points
                    case default                                        ! don't change indices
                        int_facs = 0.5_dp*[1,2]
                        int_dims(1,:) = [1,grid_x%n(1),grid_x%n(1)-1]   ! region 1: first and last points
                        int_dims(2,:) = [2,grid_x%n(1)-1,1]             ! region 2: intermediate points
                end select
            case (2)                                                    ! Simpson's 3/8 rule
                select case (prev_style_loc)
                    case (2,3)                                          ! change indices
                        int_facs = 0.375_dp*[3,2,3]
                        int_dims(1,:) = [1,grid_x%n(1)-2,3]             ! region 1: intermediate points ~ 3
                        int_dims(2,:) = [2,grid_x%n(1)-1,3]             ! region 2: intermediate points ~ 2
                        int_dims(3,:) = [3,grid_x%n(1),3]               ! region 3: intermediate points ~ 3
                    case default                                        ! don't change indices
                        int_facs = 0.375_dp*[1,3,3,2]
                        int_dims(1,:) = [1,grid_x%n(1),grid_x%n(1)-1]   ! region 1: first and last points
                        int_dims(2,:) = [2,grid_x%n(1)-2,3]             ! region 2: intermediate points ~ 3
                        int_dims(3,:) = [3,grid_x%n(1)-1,3]             ! region 3: intermediate points ~ 3
                        int_dims(4,:) = [4,grid_x%n(1)-3,3]             ! region 4: intermediate points ~ 2
                end select
        end select

        ! set up multiplicative factor for previous level
        select case (prev_style_loc)
            case (1,3)                                                  ! add to integral of previous equilibrium job
                prev_mult_fac = 1.0_dp
            case (2)                                                    ! add to integral of previous Richardson level / 2
                prev_mult_fac = 0.5_dp
            case default                                                ! don't add, overwrite
                prev_mult_fac = 0.0_dp
        end select

        ! set up step size
        allocate(step_size(grid_x%n(2),grid_x%loc_n_r))
        step_size = ang_par_f(2,:,:)-ang_par_f(1,:,:)
        if (rich_lvl.gt.1) step_size = step_size*0.5_dp                 ! only half of points present: step size is actually half

        ! initialize local V_int
        allocate(v_int_work(grid_x%n(2),grid_x%loc_n_r))

        ! loop over all modes
        do m = 1,x%n_mod(2)
            do k = 1,x%n_mod(1)
                ! check whether mode combination needs to be calculated
                calc_this(1) = is_necessary_x(.true.,[k,m],lim_sec_x)
                calc_this(2) = is_necessary_x(.false.,[k,m],lim_sec_x)

```

```
                ! set up local and total indices
                c_loc(1) = c([k,m],.true.,n_mod_x,lim_sec_x)
                c_loc(2) = c([k,m],.false.,n_mod_x,lim_sec_x)
                c_tot(1) = c([lim_sec_x(1,1)-1+k,lim_sec_x(1,2)-1+m],.true.,&
                    &n_mod_x)
                c_tot(2) = c([lim_sec_x(1,1)-1+k,lim_sec_x(1,2)-1+m],.false.,&
                    &n_mod_x)

                ! calculate J_exp_ang
                do kd = 1,grid_x%loc_n_r
                    if (use_pol_flux_f) then
                        j_exp_ang(:,:,kd) = j(:,:,kd)*&
                            &exp(iu*(x%m_1(kd,k)-x%m_2(kd,m))*&
                            &ang_par_f(:,:,kd))
                    else
                        j_exp_ang(:,:,kd) = j(:,:,kd)*&
                            &exp(-iu*(x%n_1(kd,k)-x%n_2(kd,m))*&
                            &ang_par_f(:,:,kd))
                    end if
                end do

                ! integrate PV_0 and KV_0
                if (calc_this(1)) then
                    x_int%PV_0(1,:,:,c_tot(1)) = &
                        &prev_mult_fac*x_int%PV_0(1,:,:,c_tot(1))
                    x_int%KV_0(1,:,:,c_tot(1)) = &
                        &prev_mult_fac*x_int%KV_0(1,:,:,c_tot(1))
                    call calc_magn_int_loc(x%PV_0(:,:,:,c_loc(1))*alpha_fac,&
                        &x_int%PV_0(1,:,:,c_tot(1)),v_int_work,step_size)
                    call calc_magn_int_loc(x%KV_0(:,:,:,c_loc(1))*alpha_fac,&
                        &x_int%KV_0(1,:,:,c_tot(1)),v_int_work,step_size)
                end if

                ! integrate PV_1 and KV_1
                if (calc_this(2)) then
                    x_int%PV_1(1,:,:,c_tot(2)) = &
                        &prev_mult_fac*x_int%PV_1(1,:,:,c_tot(2))
                    x_int%KV_1(1,:,:,c_tot(2)) = &
                        &prev_mult_fac*x_int%KV_1(1,:,:,c_tot(2))
                    call calc_magn_int_loc(x%PV_1(:,:,:,c_loc(2))*alpha_fac,&
                        &x_int%PV_1(1,:,:,c_tot(2)),v_int_work,step_size)
                    call calc_magn_int_loc(x%KV_1(:,:,:,c_loc(2))*alpha_fac,&
                        &x_int%KV_1(1,:,:,c_tot(2)),v_int_work,step_size)
                end if

                ! integrate PV_2 and KV_2
                if (calc_this(1)) then
                    x_int%PV_2(1,:,:,c_tot(1)) = &
                        &prev_mult_fac*x_int%PV_2(1,:,:,c_tot(1))
                    x_int%KV_2(1,:,:,c_tot(1)) = &
                        &prev_mult_fac*x_int%KV_2(1,:,:,c_tot(1))
                    call calc_magn_int_loc(x%PV_2(:,:,:,c_loc(1))*alpha_fac,&
                        &x_int%PV_2(1,:,:,c_tot(1)),v_int_work,step_size)
                    call calc_magn_int_loc(x%KV_2(:,:,:,c_loc(1))*alpha_fac,&
                        &x_int%KV_2(1,:,:,c_tot(1)),v_int_work,step_size)
                end if
            end do
        end do

        ! clean up
        nullify(ang_par_f)
        select case (x_grid_style)
            case (1)                                                    ! equilibrium
                ! do nothing
            case (2,3)                                                  ! solution or enriched
                deallocate(j)
        end select
        nullify(j)

        ! user output
        call lvl_ud(-1)
    contains
        ! Integrate local magnetic integral, adding to the previous result.
        !
        ! Makes use of nr_int_regions, int_dims, int_facs and J_exp_ang.
        !> \private
        subroutine calc_magn_int_loc(V,V_int,V_int_work,step_size)
            ! input / output
            complex(dp), intent(in) :: v(:,:,:)                         ! V
            complex(dp), intent(inout) :: v_int(:,:)                    ! integrated V
            complex(dp), intent(inout) :: v_int_work(:,:)               ! workint variable
            real(dp) :: step_size(:,:)                                  ! step size for every normal point

            ! local variables
            integer :: id, kd, ld                                       ! counters

            ! loop over regions
```

```
            do ld = 1,nr_int_regions
                ! initialize
                v_int_work = 0

                ! integrate
                do kd = 1,size(step_size,2)
                    do id = int_dims(ld,1),int_dims(ld,2),int_dims(ld,3)
                        v_int_work(:,kd) = v_int_work(:,kd) + &
                            &j_exp_ang(id,:,kd)*v(id,:,kd)*step_size(:,kd)
                    end do
                end do

                ! scale by integration factor and update V_int
                v_int = v_int + v_int_work*int_facs(ld)
            end do
        end subroutine calc_magn_int_loc
    end function calc_magn_ints

    !> Divides the perturbation jobs.
    !!
    !! This  concerns the  calculation of  the magnetic  integrals of  blocks of
    !! tensorial perturbation variables. These are  set up using the equilibrium
    !! and vectorial perturbation variables that  are stored in memory, but only
    !! the  integrated tensorial  result is  stored in  memory, with  negligible
    !! variables size.
    !!
    !! The size of the (k,m) pairs to  be calculated is determined by looking at
    !! what  fits in  memory  when the  equilibrium  and vectorial  perturbation
    !! variables are stored in memory first.
    !!
    !! \return ierr
    integer function divide_x_jobs(arr_size) result(ierr)
        use num_vars, only: max_x_mem, n_procs, x_jobs_lims
        use x_vars, only: n_mod_x
        use x_utilities, only: calc_memory_x

        character(*), parameter :: rout_name = 'divide_X_jobs'

        ! input / output
        integer, intent(in) :: arr_size                                         !< array size (using loc_n_r)

        ! local variables
        integer :: n_div                                                        ! factor by which to divide the total size
        real(dp) :: mem_size                                                    ! approximation of memory required for X
variables
        integer :: n_mod_block                                                  ! nr. of modes in block
        integer, allocatable :: n_mod_loc(:)                                    ! number of modes per block
        character(len=max_str_ln) :: block_message                              ! message about how many blocks
        character(len=max_str_ln) :: err_msg                                    ! error message

        ! initialize ierr
        ierr = 0

        ! user output
        call writo('Dividing the perturbation jobs')
        call lvl_ud(1)

        ! calculate largest possible block of (k,m) values
        n_div = 0
        mem_size = huge(1._dp)
        do while (mem_size.gt.max_x_mem)
            n_div = n_div + 1
            n_mod_block = ceiling(n_mod_x*1._dp/n_div)
            ierr = calc_memory_x(2,arr_size,n_mod_block,mem_size)
            chckerr('')
            if (n_div.gt.n_mod_x) then
                ierr = 1
                err_msg = 'The memory limit is too low'
                chckerr(err_msg)
            end if
        end do
        if (n_div.gt.1) then
            block_message = 'The '//trim(i2str(n_mod_x))//&
                &' Fourier modes are split into '//trim(i2str(n_div))//&
                &' and '//trim(i2str(n_div**2))//' jobs are done separately'
            if (n_procs.lt.n_div**2) then
                block_message = trim(block_message)//', '//&
                    &trim(i2str(n_procs))//' at a time'
            else
                block_message = trim(block_message)//', but simultaneously'
            end if
        else
            block_message = 'The '//trim(i2str(n_mod_x))//' Fourier modes &
                &can be done without splitting them'
        end if
        call writo(block_message)
        call writo('The total memory for all processes together is estimated &
```

```
            &to be about '//trim(i2str(ceiling(mem_size)))//'MB')
        call lvl_ud(1)
        call writo('(maximum: '//trim(i2str(ceiling(max_x_mem)))//&
            &'MB left over from equilibrium variables)')
        call lvl_ud(-1)

        ! set up jobs data as illustrated below for 3 divisions, order 1:
        !   [1,2,3]
        ! or order 2:
        !   [1,4,7]
        !   [2,5,8]
        !   [3,6,9]
        ! etc.
        ! Also initialize the jobs taken to false.
        allocate(n_mod_loc(n_div))
        n_mod_loc = n_mod_x/n_div                                       ! number of radial points on this processor
        n_mod_loc(1:mod(n_mod_x,n_div)) = n_mod_loc(1:mod(n_mod_x,n_div)) + 1   ! add a mode to if there is a remainder
        call calc_x_jobs_lims(x_jobs_lims,n_mod_loc,2)

        ! user output
        call lvl_ud(-1)
        call writo('Perturbation jobs divided')
    contains
        ! Calculate X_jobs_lims.
        !
        ! These are  stored as follows: The  job index is the  second one, while
        ! the first index  ranges from 1..2. For  every job in the  range of job
        ! indices, thee first index shows the limits of
        ! the different orders sequentially. For example:
        !   - for order 1: [3,4],
        !       which corresponds to the 1-D range 3..4,
        !   - for order 2: [3,4,1,5],
        !       which corresponds to the 2-D range 3..4 x 1..5,
        !
        ! etc.
        !> \private
        recursive subroutine calc_x_jobs_lims(res,n_mod,ord)
            ! input / output
            integer, intent(inout), allocatable :: res(:,:)             ! result
            integer, intent(in) :: n_mod(:)                             ! X jobs data
            integer, intent(in) :: ord                                  ! order of data

            ! local variables
            integer, allocatable :: res_loc(:,:)                        ! local result
            integer :: n_div                                            ! nr. of divisions of modes
            integer :: id                                               ! counter

            ! set up nr. of divisions
            n_div = size(n_mod)

            ! (re)allocate result
            if (allocated(res)) deallocate(res)
            allocate(res(2*ord,n_div**ord))

            ! loop over divisions
            do id = 1,n_div
                if (ord.gt.1) then                                      ! call lower order
                    call calc_x_jobs_lims(res_loc,n_mod,ord-1)
                    res(1:2*ord-2,(id-1)*n_div**(ord-1)+1:id*n_div**(ord-1)) = &
                        &res_loc
                end if                                                  ! set for own order
                res(2*ord-1,(id-1)*n_div**(ord-1)+1:id*n_div**(ord-1)) = &
                    &sum(n_mod(1:id-1))+1
                res(2*ord,(id-1)*n_div**(ord-1)+1:id*n_div**(ord-1)) = &
                    &sum(n_mod(1:id))
            end do
        end subroutine calc_x_jobs_lims
    end function divide_x_jobs

#if ldebug
    !> Prints debug information for X_1 driver
    integer function print_debug_x_1(mds,grid_X,X_1) result(ierr)
        use num_vars, only: use_pol_flux_f
        use grid_utilities, only: trim_grid
        use grid_vars, only: alpha, n_alpha
        use eq_vars, only: max_flux_f
        use rich_vars, only: rich_lvl

        character(*), parameter :: rout_name = 'print_debug_X_1'

        ! input / output
        type(modes_type), intent(in) :: mds                             !< general modes variables
        type(grid_type), intent(in) :: grid_x                           !< perturbation grid
        type(x_1_type), intent(in) :: x_1                               !< vectorial X variables

        ! local variabbles
        type(grid_type) :: grid_trim                                    ! trimmed grid
```

```
        character(len=max_str_ln), allocatable :: var_names(:)                  ! names of variables
        character(len=max_str_ln) :: file_name                                  ! name of file
        integer :: k                                                            ! local row index in flux surface
        integer :: ld, kd, jd, rd, id                                           ! counters
        integer :: min_nm_x                                                     ! minimal n (tor. flux) or m (pol. flux)
        integer :: n_mod_tot                                                    ! total number of modes
        integer :: kdl_tot(2)                                                   ! limits on total normal index for a mode
        integer :: rdl                                                          ! rd in local tables
        integer :: kd_loc                                                       ! local kd
        integer :: kd_loc_trim                                                  ! local kd on trimmed grid
        integer :: plot_dim(4)                                                  ! dimensions of plot
        integer :: plot_offset(4)                                               ! local offset of plot
        complex(dp), allocatable :: u(:,:,:,:,:)                                ! U_i for all modes
        complex(dp), allocatable :: du(:,:,:,:,:)                               ! DU_i for all modes
        real(dp), allocatable :: x_plot(:,:,:,:)                                ! X of plot
        real(dp), allocatable :: y_plot(:,:,:,:)                                ! Y of plot
        real(dp), allocatable :: z_plot(:,:,:,:)                                ! Y of plot

        ! initiaiize ierr
        ierr = 0

        ! trim grid
        ierr = trim_grid(grid_x,grid_trim)
        chckerr('')

        ! set local n_mod and allocate integrated quantities
        min_nm_x = minval(mds%sec(:,1),1)
        n_mod_tot = size(mds%sec,1)
        allocate(u(n_mod_tot,grid_trim%n(1),grid_trim%loc_n_r,grid_trim%n(2),&
            &0:1))
        allocate(du(n_mod_tot,grid_trim%n(1),grid_trim%loc_n_r,grid_trim%n(2),&
            &0:1))
        allocate(x_plot(n_mod_tot,grid_trim%n(1),grid_trim%loc_n_r,&
            &grid_trim%n(2)))
        allocate(y_plot(n_mod_tot,grid_trim%n(1),grid_trim%loc_n_r,&
            &grid_trim%n(2)))
        allocate(z_plot(n_mod_tot,grid_trim%n(1),grid_trim%loc_n_r,&
            &grid_trim%n(2)))
        u = 0._dp
        du = 0._dp

        ! setup X, Y and Z of plot
        do kd = 1,grid_trim%loc_n_r
            z_plot(:,:,kd,1) = 1000*grid_trim%loc_r_F(kd)/max_flux_f*2*pi
            do rd = 1,n_mod_tot
                do id = 1,grid_trim%n(1)
                    x_plot(rd,id,kd,:) = min_nm_x+rd-1
                    do jd = 1,grid_trim%n(2)
                        if (use_pol_flux_f) then
                            y_plot(rd,id,kd,jd) = grid_trim%theta_F(id,jd,kd)*10
                        else
                            y_plot(rd,id,kd,jd) = grid_trim%zeta_F(id,jd,kd)*10
                        end if
                    end do
                end do
            end do
        end do

        ! select all modes combinations
        do k = 1,size(mds%sec,1)
            ! set normal limits for mode k
            kdl_tot(1) = mds%sec(k,2)
            kdl_tot(2) = mds%sec(k,3)

            ! limit to trimmed grid range
            kdl_tot(1) = max(kdl_tot(1),grid_trim%i_min)
            kdl_tot(2) = min(kdl_tot(2),grid_trim%i_max)

            ! skip if out of normal range
            if (kdl_tot(1).gt.kdl_tot(2)) cycle

            ! total mode index (starting from 1)
            rd = mds%sec(k,1)-min_nm_x+1

            ! indices in local tables
            rdl = mds%sec(k,4)

            ! loop over all normal points
            do kd = kdl_tot(1),kdl_tot(2)
                ! set indices
                kd_loc = kd - grid_x%i_min + 1
                kd_loc_trim = kd - grid_trim%i_min + 1

                u(rd,:,kd_loc_trim,:,0) = x_1%U_0(:,:,kd_loc,rdl)
                u(rd,:,kd_loc_trim,:,1) = x_1%U_1(:,:,kd_loc,rdl)
                du(rd,:,kd_loc_trim,:,0) = x_1%DU_0(:,:,kd_loc,rdl)
                du(rd,:,kd_loc_trim,:,1) = x_1%DU_1(:,:,kd_loc,rdl)
```

```
            end do
        end do

        ! plot
        allocate(var_names(n_alpha))
        do ld = 1,size(var_names)
            var_names(ld) = 'alpha = '//trim(r2strt(alpha(ld)))
        end do
        plot_dim = [n_mod_tot,grid_trim%n(1),grid_trim%n(3),n_alpha]
        plot_offset = [0,0,grid_trim%i_min-1,0]
        do id = 0,1
            file_name = 'U_'//trim(i2str(id))//'_R'//&
                &trim(i2str(rich_lvl))
            call plot_hdf5(var_names,'RE_'//trim(file_name),&
                &rp(u(:,:,:,:,id)),x=x_plot,y=y_plot,z=z_plot,&
                &tot_dim=plot_dim, loc_offset=plot_offset,&
                &col_id=4,col=1)
            call plot_hdf5(var_names,'IM_'//trim(file_name),&
                &ip(u(:,:,:,:,id)),x=x_plot,y=y_plot,z=z_plot,&
                &tot_dim=plot_dim, loc_offset=plot_offset,&
                &col_id=4,col=1)

            file_name = 'DU_'//trim(i2str(id))//'_R'//&
                &trim(i2str(rich_lvl))
            call plot_hdf5(var_names,'RE_'//trim(file_name),&
                &rp(du(:,:,:,:,id)),x=x_plot,y=y_plot,z=z_plot,&
                &tot_dim=plot_dim, loc_offset=plot_offset,&
                &col_id=4,col=1)
            call plot_hdf5(var_names,'IM_'//trim(file_name),&
                &ip(du(:,:,:,:,id)),x=x_plot,y=y_plot,z=z_plot,&
                &tot_dim=plot_dim, loc_offset=plot_offset,&
                &col_id=4,col=1)
        end do
        deallocate(var_names)

        ! clean up
        call grid_trim%dealloc()
    end function print_debug_x_1

    !> Prints debug information for X_2 driver
    integer function print_debug_x_2(mds,grid_X,X_2_int) result(ierr)
        use num_utilities, only: c, con
        use x_vars, only: n_mod_x
        use grid_vars, only: alpha, n_alpha
        use grid_utilities, only: trim_grid
        use eq_vars, only: max_flux_f
        use rich_vars, only: rich_lvl

        character(*), parameter :: rout_name = 'print_debug_X_2'

        ! input / output
        type(modes_type), intent(in) :: mds                                 !< general modes variables
        type(grid_type), intent(in) :: grid_x                               !< perturbation grid
        type(x_2_type), intent(in) :: x_2_int                               !< tensorial X variables

        ! local variables
        type(grid_type) :: grid_trim                                        ! trimmed grid
        character(len=max_str_ln), allocatable :: var_names(:)              ! names of variables
        character(len=max_str_ln) :: file_name                              ! name of file
        integer :: k, m                                                     ! local row and column index in flux surface
        integer :: id, ld, kd, rd, cd                                       ! counters
        integer :: min_nm_x                                                 ! minimal n (tor. flux) or m (pol. flux)
        integer :: n_mod_tot                                                ! total number of modes
        integer :: kdl_tot(2)                                               ! limits on total normal index for a mode
combination
        integer :: kd_loc                                                   ! local kd
        integer :: kd_loc_trim                                              ! local kd on trimmed grid
        integer :: plot_dim(4)                                              ! dimensions of plot
        integer :: plot_offset(4)                                           ! local offset of plot
        integer :: c_loc(2)                                                 ! table index for symmetric and asymmetric
quantities
        integer :: rdl, cdl                                                 ! rd and cd in local tables
        complex(dp), allocatable :: pv_int(:,:,:,:,:)                       ! integrated PV_i for all mode combinations
        complex(dp), allocatable :: kv_int(:,:,:,:,:)                       ! integrated KV_i for all mode combinations
        real(dp), allocatable :: x_plot(:,:,:,:)                            ! X of plot
        real(dp), allocatable :: y_plot(:,:,:,:)                            ! Y of plot
        real(dp), allocatable :: z_plot(:,:,:,:)                            ! Y of plot

        ! initiaiize ierr
        ierr = 0

        ! trim grid
        ierr = trim_grid(grid_x,grid_trim)
        chckerr('')

        ! set local n_mod and allocate integrated quantities
        min_nm_x = minval(mds%sec(:,1),1)
```

```fortran
        n_mod_tot = size(mds%sec,1)
        allocate(pv_int(n_mod_tot,n_mod_tot,grid_trim%loc_n_r,&
            &grid_trim%n(2),0:2))
        allocate(kv_int(n_mod_tot,n_mod_tot,grid_trim%loc_n_r,&
            &grid_trim%n(2),0:2))
        allocate(x_plot(n_mod_tot,n_mod_tot,grid_trim%loc_n_r,1))
        allocate(y_plot(n_mod_tot,n_mod_tot,grid_trim%loc_n_r,1))
        allocate(z_plot(n_mod_tot,n_mod_tot,grid_trim%loc_n_r,1))
        pv_int = 0._dp
        kv_int = 0._dp

        ! setup X, Y and Z of plot
        do kd = 1,grid_trim%loc_n_r
            z_plot(:,:,kd,1) = 1000*grid_trim%loc_r_F(kd)/max_flux_f*2*pi
            do cd = 1,n_mod_tot
                do rd = 1,n_mod_tot
                    x_plot(rd,cd,kd,1) = min_nm_x+rd-1
                    y_plot(rd,cd,kd,1) = min_nm_x+cd-1
                end do
            end do
        end do

        ! select all mode combinations
        do m = 1,size(mds%sec,1)
            do k = 1,size(mds%sec,1)
                ! set normal limits for mode pair (k,m)
                kdl_tot(1) = max(mds%sec(k,2),mds%sec(m,2))
                kdl_tot(2) = min(mds%sec(k,3),mds%sec(m,3))

                ! limit to trimmed grid range
                kdl_tot(1) = max(kdl_tot(1),grid_trim%i_min)
                kdl_tot(2) = min(kdl_tot(2),grid_trim%i_max)

                ! skip if out of normal range
                if (kdl_tot(1).gt.kdl_tot(2)) cycle

                ! total mode combinations indices (starting from 1)
                rd = mds%sec(k,1)-min_nm_x+1
                cd = mds%sec(m,1)-min_nm_x+1

                ! indices in local tables
                rdl = mds%sec(k,4)
                cdl = mds%sec(m,4)

                ! index in tables
                c_loc(1) = c([rdl,cdl],.true.,n_mod_x)
                c_loc(2) = c([rdl,cdl],.false.,n_mod_x)

                ! loop over all normal points
                do kd = kdl_tot(1),kdl_tot(2)
                    ! set indices
                    kd_loc = kd - grid_x%i_min + 1
                    kd_loc_trim = kd - grid_trim%i_min + 1

                    ! symmetric quantities
                    pv_int(rd,cd,kd_loc_trim,:,0) = &
                        &con(x_2_int%PV_0(1,:,kd_loc,c_loc(1)),&
                        &[rdl,cdl],.true.,[n_alpha])
                    pv_int(rd,cd,kd_loc_trim,:,2) = &
                        &con(x_2_int%PV_2(1,:,kd_loc,c_loc(1)),&
                        &[rdl,cdl],.true.,[n_alpha])
                    kv_int(rd,cd,kd_loc_trim,:,0) = &
                        &con(x_2_int%KV_0(1,:,kd_loc,c_loc(1)),&
                        &[rdl,cdl],.true.,[n_alpha])
                    kv_int(rd,cd,kd_loc_trim,:,2) = &
                        &con(x_2_int%KV_2(1,:,kd_loc,c_loc(1)),&
                        &[rdl,cdl],.true.,[n_alpha])

                    ! asymmetric quantities
                    pv_int(rd,cd,kd_loc_trim,:,1) = &
                        &x_2_int%PV_1(1,:,kd_loc,c_loc(2))
                    kv_int(rd,cd,kd_loc_trim,:,1) = &
                        &x_2_int%KV_1(1,:,kd_loc,c_loc(2))
                end do
            end do
        end do

        ! plot
        allocate(var_names(n_alpha))
        do ld = 1,size(var_names)
            var_names(ld) = 'alpha = '//trim(r2strt(alpha(ld)))
        end do
        plot_dim = [n_mod_tot,n_mod_tot,grid_trim%n(3),n_alpha]
        plot_offset = [0,0,grid_trim%i_min-1,0]
        do id = 0,2
            file_name = 'PV_'//trim(i2str(id))//'_int_R'//&
                &trim(i2str(rich_lvl))
```

```
            call plot_hdf5(var_names,'RE_'//trim(file_name),&
                &rp(pv_int(:,:,:,:,id)),x=x_plot,y=y_plot,z=z_plot,&
                &tot_dim=plot_dim, loc_offset=plot_offset,&
                &col_id=4,col=1)
            call plot_hdf5(var_names,'IM_'//trim(file_name),&
                &ip(pv_int(:,:,:,:,id)),x=x_plot,y=y_plot,z=z_plot,&
                &tot_dim=plot_dim, loc_offset=plot_offset,&
                &col_id=4,col=1)

            file_name = 'KV_'//trim(i2str(id))//'_int_R'//&
                &trim(i2str(rich_lvl))
            call plot_hdf5(var_names,'RE_'//trim(file_name),&
                &rp(kv_int(:,:,:,:,id)),x=x_plot,y=y_plot,z=z_plot,&
                &tot_dim=plot_dim, loc_offset=plot_offset,&
                &col_id=4,col=1)
            call plot_hdf5(var_names,'IM_'//trim(file_name),&
                &ip(kv_int(:,:,:,:,id)),x=x_plot,y=y_plot,z=z_plot,&
                &tot_dim=plot_dim, loc_offset=plot_offset,&
                &col_id=4,col=1)
        end do
        deallocate(var_names)

        ! clean up
        call grid_trim%dealloc()
    end function print_debug_x_2
#endif
end module
```

D.100 /Users/toonweyens/Code/PB3D/Modules/X_utilities.f90 File Reference

Interfaces and Types

- interface x_utilities::sec_ind_loc2tot

  Returns the `sec_ind_tot` used to refer to a perturbation quantity.

Modules

- module x_utilities

  Numerical utilities related to perturbation operations.

Functions/Subroutines

- subroutine, public x_utilities::get_sec_x_range (sec_x_range_loc, sec_x_range_tot, m, sym, lim_sec_x)

  Gets one of the the local ranges of contiguous tensorial perturbation variables to be printed or read during one call of the corresponding HDF5 variables.
- logical function, public x_utilities::do_x ()

  Tests whether this perturbatino job should be done.
- logical function, public x_utilities::is_necessary_x (sym, sec_x_id, lim_sec_x)

  Determines whether a variable needs to be considered:
- integer function, public x_utilities::calc_memory_x (ord, arr_size, n_mod, mem_size)

  Calculate memory in MB necessary for X variables.
- integer function, public x_utilities::trim_modes (mds_i, mds_o, id_lim_i, id_lim_o)

  Limit input mode range to output mode range.

D.101 X_utilities.f90

Go to the documentation of this file.

```
!-----------------------------------------------------------------------------!
!> Numerical utilities related to perturbation operations.
!-----------------------------------------------------------------------------!
module x_utilities
#include <PB3D_macros.h>
    use str_utilities
    use messages
    use num_vars, only: dp, max_name_ln, iu, max_str_ln
    use grid_vars, only: grid_type
    use x_vars, only: n_mod_x

    implicit none

    private
    public sec_ind_loc2tot, is_necessary_x, get_sec_x_range, calc_memory_x, &
        &do_x, trim_modes

    ! interfaces

    !> \public Returns the \c sec_ind_tot used to refer to a perturbation
    !! quantity.
    interface sec_ind_loc2tot
        !> \public
        module procedure sec_ind_loc2tot_1
        !> \public
        module procedure sec_ind_loc2tot_2
    end interface

contains
    !> \private vectorial version
    function sec_ind_loc2tot_1(id,lim_sec_X) result(res)
        ! input / output
        integer, intent(in) :: id                                       !< mode index
        integer, intent(in), optional :: lim_sec_x(2)                   !< limits of \c m_X (pol. flux) or \c n_X (tor. flux)
        integer :: res                                                  !< output

        ! local variables
        integer :: lim_sec_x_loc(2)                                     ! local lim_sec_X

        ! set local lim_sec_X
        lim_sec_x_loc = [1,n_mod_x]
        if (present(lim_sec_x)) lim_sec_x_loc = lim_sec_x

        ! set sec_ind_tot
        res = lim_sec_x_loc(1)-1+id
    end function sec_ind_loc2tot_1
    !> \private tensorial version
    function sec_ind_loc2tot_2(id,jd,lim_sec_X) result(res)
        ! input / output
        integer, intent(in) :: id                                       !< mode index for dimension 1
        integer, intent(in) :: jd                                       !< mode index for dimension 1
        integer, intent(in), optional :: lim_sec_x(2,2)                 !< limits of \c m_X (pol flux) or \c n_X (tor flux) for both dimensions
        integer :: res(2)                                               !< output

        ! local variables
        integer :: lim_sec_x_loc(2,2)                                   ! local lim_sec_X

        ! set local lim_sec_X
        lim_sec_x_loc(:,1) = [1,n_mod_x]
        lim_sec_x_loc(:,2) = [1,n_mod_x]
        if (present(lim_sec_x)) lim_sec_x_loc = lim_sec_x

        ! set sec_ind_tot
        res = [lim_sec_x_loc(1,1)-1+id,lim_sec_x_loc(1,2)-1+jd]
    end function sec_ind_loc2tot_2

    !> Gets one of the the local ranges of contiguous tensorial perturbation
    !! variables to be printed or read during one call of the corresponding HDF5
    !! variables.
    !!
    !! More specifically, a range of indices \c k in the first dimension is
    !! given for every value of the indices \c m in the second dimension.
    !!
    !! An example is now given for the subrange <tt>[2:3,2:5]</tt> of a the
    !! total range <tt>[1:5, 1:5]</tt>. For asymmetric variables the situation
    !! is simple: The \c k range is <tt>[2:3]</tt> for all 5 values of \c m.
    !! However, for symmetric variables, the upper diagonal values are not
    !! stored, which gives \c k ranges <tt>[2:3]</tt>, <tt>[3:3]</tt> and no
    !! range for \c m = 4 and 5.
```

```
    !!
    !! This routine then translates these  ranges to the corresponding 1-D ranges
    !! that  are used  in  the actual  variables. For  above  example, the  total
    !! indices are
    !!  \f[
    !!      \left(\begin{array}{ccccc}
    !!          1 &  6 & 11 & 16 & 21 \\
    !!          2 &  7 & 12 & 17 & 22 \\
    !!          3 &  8 & 13 & 18 & 23 \\
    !!          4 &  9 & 14 & 19 & 24 \\
    !!          5 & 10 & 15 & 20 & 25
    !!      \end{array}\right)
    !!      \rightarrow \left[7:8\right], \left[12:13\right],
    !!      \left[17:18\right] \ \text{and} \ \left[22:23\right],
    !!  \f]
    !! for asymmetric variables and
    !!  \f[
    !!      \left(\begin{array}{ccccc}
    !!          1 & 2 &  3 &  4 &  5 \\
    !!          2 & 6 &  7 &  8 &  9 \\
    !!          3 & 7 & 10 & 11 & 12 \\
    !!          4 & 8 & 11 & 13 & 14 \\
    !!          5 & 9 & 12 & 14 & 15
    !!      \end{array}\right)
    !!      \rightarrow \left[6:7\right], \left[10:10\right],
    !!      \left[:\right] \ \text{and} \ \left[:\right],
    !!  \f]
    !! for symmetric variables.
    !!
    !! These can  then related  to the  local indices for  the variables  in this
    !! perturbation job.
    !!
    !! For above example, the results are:
    !!
    !!   <tt>[1:2], [3:4], [5:6] and [7:8]</tt>,
    !!
    !! for asymmetric variables and
    !!
    !!   <tt>[1:2], [3:3], [:] and [:]</tt>,
    !!
    !! for symmetric variables.
    !!
    !! As can  be seen, the  local ranges of the  variables in the  submatrix of
    !! this  perturbation job  are (designed  to be)  contiguous, but  the total
    !! ranges of the variables in the submatrix are clearly not in general.
    !!
    !! The procedure outputs both the local and total ranges.
    subroutine get_sec_x_range(sec_X_range_loc,sec_X_range_tot,m,sym,lim_sec_X)
        use x_vars, only: n_mod_x
        use num_utilities, only: c

        ! input / output
        integer, intent(inout) :: sec_x_range_loc(2)                          !< start and end of local range in dimension 1 (vertical)
        integer, intent(inout) :: sec_x_range_tot(2)                          !< start and end of total range in dimension 1 (vertical)
        integer, intent(in) :: m                                              !< dimension 2 (horizontal)
        logical, intent(in) :: sym                                            !< whether the variable is symmetric
        integer, intent(in), optional :: lim_sec_x(2,2)                       !< limits of \c m_X (pol. flux) or \c n_X (tor. flux)

        ! local variables
        integer :: k_range_loc(2)                                             ! local range of k
        integer :: k                                                          ! counter
        integer :: n_mod                                                      ! local n_mod

        ! set number of modes of dimension 1
        if (present(lim_sec_x)) then
            n_mod = lim_sec_x(2,1)-lim_sec_x(1,1)+1
        else
            n_mod = n_mod_x
        endif

        ! initialize secondary X range
        k_range_loc = [n_mod+1,0]                                             ! initialize to inverted values, out of bounds

        ! find start and end of k range
        find_start: do k = 1,n_mod
            if (is_necessary_x(sym,[k,m],lim_sec_x)) then                     ! first necessary pair [k,m]
                k_range_loc(1) = k
                exit find_start
            end if
        end do find_start
        find_stop: do k = n_mod,k_range_loc(1),-1
            if (is_necessary_x(sym,[k,m],lim_sec_x)) then                     ! last necessary pair [k,m]
                k_range_loc(2) = k
```

```
                exit find_stop
            end if
        end do find_stop

        ! translate k range and m value to 1-D index
        do k = 1,2
            sec_x_range_loc(k) = c([k_range_loc(k),m],sym,n_mod_x,lim_sec_x)
            sec_x_range_tot(k) = c(sec_ind_loc2tot(k_range_loc(k),m,lim_sec_x),&
                &sym,n_mod_x)
        end do
    end subroutine get_sec_x_range

    !> Tests whether this perturbatino job should be done.
    !!
    !! Also increments \c X_job_nr
    logical function do_x()
        use num_vars, only: x_jobs_lims, x_job_nr

        ! increment perturbation job nr.
        x_job_nr = x_job_nr + 1

        if (x_job_nr.le.size(x_jobs_lims,2)) then
            do_x = .true.
        else
            do_x = .false.
        end if
    end function do_x

    !> Determines whether a variable needs to be  considered:
    !!
    !! This depends on whether the quantity is symmetric or not:
    !!  - Only if it is on or below the diagonal for symmetric quantities.
    !!  - Always for asymmetric quantities
    logical function is_necessary_x(sym,sec_X_id,lim_sec_X) result(res)
        ! input / output
        logical, intent(in) :: sym                                          !< whether the variable is symmetric
        integer, intent(in) :: sec_x_id(2)                                  !< mode indices
        integer, intent(in), optional :: lim_sec_x(2,2)                     !< limits of \c m_X (pol flux) or \c n_X (tor flux) for both dimensions

        ! local variables
        integer :: lim_sec_x_loc(2,2)                                       ! local version of lim_sec_X

        ! initialize res
        res = .true.

        ! modify res depending on symmetry
        if (sym) then
            ! set local lim_sec_X
            lim_sec_x_loc(:,1) = [1,n_mod_x]
            lim_sec_x_loc(:,2) = [1,n_mod_x]
            if (present(lim_sec_x)) lim_sec_x_loc = lim_sec_x

            ! set res
            if (lim_sec_x_loc(1,1)+sec_x_id(1).lt.&
                &lim_sec_x_loc(1,2)+sec_x_id(2)) res = .false.
        end if
    end function is_necessary_x

    !> Calculate memory in MB necessary for X variables.
    !!
    !! This depends on the order:
    !!  - order 1:
    !!      - <tt>4x n_par_X x n_geo x loc_n_r x n_mod</tt>
    !!  - order 2:
    !!      - <tt>2x n_par_X x n_geo x loc_n_r x n_mod^2</tt>
    !!      - <tt>4x n_par_X x n_geo x loc_n_r x n_mod(n_mod+1)/2</tt>
    !!  - higher order: not used
    !!
    !! where <tt>n_par_X x n_geo x loc_n_r</tt>  should be passed as \c arr_size
    !! and \c n_mod as well.
    !!
    !! \return ierr
    integer function calc_memory_x(ord,arr_size,n_mod,mem_size) result(ierr)
        use iso_c_binding

        character(*), parameter :: rout_name = 'calc_memory_X'

        ! input / output
        integer, intent(in) :: ord                                          !< order of data
        integer, intent(in) :: arr_size                                     !< size of part of X array
        integer, intent(in) :: n_mod                                        !< number of modes
        real(dp), intent(inout) :: mem_size                                 !< total size

        ! local variables
        integer(C_SIZE_T) :: dp_size                                        ! size of dp
        character(len=max_str_ln) :: err_msg                                ! error message
```

```fortran

        ! initialize ierr
        ierr = 0

        call lvl_ud(1)

        ! get size of complex variable
        dp_size = 2*sizeof(1._dp)                                       ! complex variable

        ! calculate memory depending on order
        select case(ord)
            case (1)                                                    ! vectorial data: U, DU
                ! set memory size
                mem_size = 4*arr_size
                mem_size = mem_size*n_mod**ord
                mem_size = mem_size*dp_size
            case (2)                                                    ! tensorial data: PV, KV
                ! set memory size
                mem_size = arr_size
                mem_size = mem_size*(2*n_mod**ord+4*n_mod*(n_mod+1)/2)
                mem_size = mem_size*dp_size
            case default
                ierr = 1
                err_msg = 'Orders > 2 are not implemented'
                chckerr(err_msg)
        end select

        ! convert B to MB
        mem_size = mem_size*1.e-6_dp

        ! apply 50% safety factor (empirical)
        mem_size = mem_size*1.5_dp

        ! test overflow
        if (mem_size.lt.0) then
            ierr = 1
            chckerr('Overflow occured')
        end if

        call lvl_ud(-1)
    end function calc_memory_x

    !> Limit input mode range to output mode range.
    integer function trim_modes(mds_i,mds_o,id_lim_i,id_lim_o) result(ierr)
        use x_vars, only: modes_type

        character(*), parameter :: rout_name = 'trim_modes'

        ! input / output
        type(modes_type), intent(in) :: mds_i                           !< general modes variables for input
        type(modes_type), intent(in) :: mds_o                           !< general modes variables for output
        integer, intent(inout) :: id_lim_i(2)                           !< limits on input modes
        integer, intent(inout) :: id_lim_o(2)                           !< limits on output modes

        ! local variables
        integer :: m                                                    ! counter
        character(len=max_str_ln) :: err_msg                            ! error message

        ! initialize ierr
        ierr = 0

        ! find limits point where input modes and output modes coincide
        ! (input grid should comprise output grid)
        id_lim_o = [1,size(mds_o%sec,1)]
        id_lim_i = [-1,-1]
        do m = 1,size(mds_i%sec,1)
            if (mds_i%sec(m,1).eq.mds_o%sec(id_lim_o(1),1)) then
                id_lim_i(1) = m
                exit
            end if
        end do
        do m = size(mds_i%sec,1),1,-1
            if (mds_i%sec(m,1).eq.mds_o%sec(id_lim_o(2),1)) then
                id_lim_i(2) = m
                exit
            end if
        end do

        ! test
        if (any(id_lim_o.le.0)) then
            ierr = 1
            err_msg = 'Cannot find limits of input modes'
            chckerr(err_msg)
        end if
    end function trim_modes
end module x_utilities
```

D.102 /Users/toonweyens/Code/PB3D/Modules/X_vars.f90 File Reference

Interfaces and Types

- type x_vars::modes_type

  mode number type
- type x_vars::x_1_type

  vectorial perturbation type
- type x_vars::x_2_type

  tensorial perturbation type
- interface x_vars::set_nm_x

  Sets n_X and m_X.

Modules

- module x_vars

  Variables pertaining to the perturbation quantities.

Functions/Subroutines

- subroutine x_vars::init_x_1 (x, mds, grid_x, lim_sec_x)

  Initializes a vectorial perturbation.
- subroutine x_vars::init_x_2 (x, mds, grid_x, lim_sec_x, is_field_averaged)

  Initializes a tensorial perturbation.
- subroutine x_vars::copy_x_1 (x_i, mds, grid_i, x_o)

  Deep copy of vectorial perturbation variables.
- subroutine x_vars::copy_x_2 (x_i, mds, grid_i, x_o)

  Deep copy of tensorial perturbation variables.
- integer function, public x_vars::set_nn_mod (sym, lim_sec_x)

  Sets number of entries for tensorial perturbation variables.
- subroutine x_vars::dealloc_mds (mds)

  Deallocates modes variables.
- subroutine x_vars::dealloc_x_1 (x)

  Deallocates vectorial perturbation variables.
- subroutine x_vars::dealloc_x_2 (x)

  Deallocates tensorial perturbation variables.

Variables

- type(modes_type), public x_vars::mds_x

  modes variables for perturbation grid
- type(modes_type), public x_vars::mds_sol

  modes variables for solution grid
- integer, public x_vars::prim_x

  n_X (pol. flux) or m_X (tor. flux)
- integer, public x_vars::min_sec_x

  m_X (pol. flux) or n_X (tor. flux) (only for X style 1)
- integer, public x_vars::max_sec_x

  m_X (pol. flux) or n_X (tor. flux) (only for\ c X style 1)
- integer, public x_vars::n_mod_x

  size of m_X (pol. flux) or n_X (tor. flux)
- integer, public x_vars::min_nm_x = 5

  minimum for the high-n theory (debable)
- integer, dimension(:), allocatable, public x_vars::min_n_x

  lowest poloidal mode number m_X, in total eq grid
- integer, dimension(:), allocatable, public x_vars::max_n_x

  highest poloidal mode number m_X, in total eq grid
- integer, dimension(:), allocatable, public x_vars::min_m_x

  lowest poloidal mode number m_X, in total eq grid
- integer, dimension(:), allocatable, public x_vars::max_m_x

  highest poloidal mode number m_X, in total eq grid
- real(dp), public x_vars::min_r_sol

  min. normal range for pert.
- real(dp), public x_vars::max_r_sol

  max. normal range for pert.
- integer, public x_vars::n_alloc_x_1s

  nr. of allocated X_1's
- integer, public x_vars::n_alloc_x_2s

  nr. of allocated X_2's

D.103 X_vars.f90

Go to the documentation of this file.

```
!-------------------------------------------------------------------------------!
!> Variables pertaining to the perturbation quantities.
!-------------------------------------------------------------------------------!
module x_vars
#include <PB3D_macros.h>
    use str_utilities
    use messages
    use num_vars, only: dp, max_name_ln, iu, weight_dp
    use grid_vars, only: grid_type

    implicit none

    private
    public set_nm_x, set_nn_mod, &
        &n_mod_x, prim_x, min_sec_x, max_sec_x, min_nm_x, min_n_x, max_n_x, &
        &min_m_x, max_m_x, min_r_sol, max_r_sol, mds_x, mds_sol
#if ldebug
    public n_alloc_x_1s, n_alloc_x_2s
#endif

    !> mode number type
```

```
    !!
    !! Type containing information about mode numbers at every flux surface:
    !!  - mode numbers: \c n, \c m,
    !!  - information of secondary modes: \c sec,
    !!      * 1: mode number
    !!      * 2: lower limit of normal range
    !!      * 3: upper limit of normal range
    !!      * 4: index in tables
    !!      where the number of modes can be bigger than the range if the safety
    !!      factor or  rotational transform is  non-monotonous so that  the same
    !!      mode number can have multiple ranges.
    !!  - Flux normal coordinate: \c r_F.
    !!
    !! These are set up in setup_nm_x().
    type, public :: modes_type
        integer, allocatable :: n(:,:)                                  !< \f$n\f$ for all modes, in total grid
        integer, allocatable :: m(:,:)                                  !< \f$m\f$ for all modes, in total grid
        integer, allocatable :: sec(:,:)                                !< \c m or \c n for all possible modes, index and limits, in total grid
    contains
        !> deallocate
        procedure :: dealloc => dealloc_mds
    end type

    !> vectorial perturbation type
    !!
    !! The arrays here are of the form:
    !!  - \c U_x_i and DU_X_i: <tt>(1:angle_1,1:angle_2,1;n_mod)</tt>
    !!
    !! \see See grid_vars.grid_type for a discussion on \c ang_1 and \c ang_2.
    type, public :: x_1_type
        integer :: n_mod                                                !< size of \f$n\f$ and \f$m\f$ (nr. of modes)
        integer :: lim_sec_x(2)                                         !< limits of \c m_X (pol. flux) or \c n_X (tor. flux)
        integer, allocatable :: n(:,:)                                  !< vector of poloidal mode numbers
        integer, allocatable :: m(:,:)                                  !< vector of poloidal mode numbers
        complex(dp), allocatable :: u_0(:,:,:,:)                        !< \f$U_m^0\f$
        complex(dp), allocatable :: u_1(:,:,:,:)                        !< \f$U_m^1\f$
        complex(dp), allocatable :: du_0(:,:,:,:)                       !< \f$\mathcal{J}\vec{B}\cdot\nabla U_m^0\f$
        complex(dp), allocatable :: du_1(:,:,:,:)                       !< \f$\mathcal{J}\vec{B}\cdot\nabla U_m^1\f$
#if ldebug
        real(dp) :: estim_mem_usage                                     !< estimated memory usage \ldebug
#endif
    contains
        !> initialize
        procedure :: init => init_x_1
        !> copy
        procedure :: copy => copy_x_1
        !> deallocate
        procedure :: dealloc => dealloc_x_1
    end type

    !> tensorial perturbation type
    !!
    !! The arrays here are of the form:
    !!  - \c PV_i and KV_i: <tt>(1:angle_1,1:angle_2,1;n_mod^2)</tt>
    !!
    !! \see See grid_vars.grid_type for a discussion on \c ang_1 and \c ang_2.
    !!
    !! \note This  type is also  used for field-averaged  tensorial perturbation
    !! variables, with \c angle_1 of size 1.
    type, public :: x_2_type
        integer :: n_mod(2)                                             !< size of \f$n\f$ and \f$m\f$ (nr. of modes)
        integer :: lim_sec_x(2,2)                                       !< limits of \c m_X (pol. flux) or \c n_X (tor. flux)
        integer, allocatable :: n_1(:,:)                                !< vector of toroidal mode numbers of dimension 1
        integer, allocatable :: n_2(:,:)                                !< vector of toroidal mode numbers of dimension 2
        integer, allocatable :: m_1(:,:)                                !< vector of poloidal mode numbers of dimension 1
        integer, allocatable :: m_2(:,:)                                !< vector of poloidal mode numbers of dimension 2
        complex(dp), allocatable :: pv_0(:,:,:,:)                       !< \f$\widetilde{PV}^0\f$ coefficient
        complex(dp), allocatable :: pv_1(:,:,:,:)                       !< \f$\widetilde{PV}^1\f$ coefficient
        complex(dp), allocatable :: pv_2(:,:,:,:)                       !< \f$\widetilde{PV}^2\f$ coefficient
        complex(dp), allocatable :: kv_0(:,:,:,:)                       !< \f$\widetilde{KV}^0\f$ coefficient
        complex(dp), allocatable :: kv_1(:,:,:,:)                       !< \f$\widetilde{KV}^1\f$ coefficient
        complex(dp), allocatable :: kv_2(:,:,:,:)                       !< \f$\widetilde{KV}^2\f$ coefficient
#if ldebug
        real(dp) :: estim_mem_usage                                     !< estimated memory usage \ldebug
#endif
    contains
        !> initialize
        procedure :: init => init_x_2
```

```
        !> copy
        procedure :: copy => copy_x_2
        !> deallocate
        procedure :: dealloc => dealloc_x_2
    end type

    ! interfaces

    !> \public Sets \c n_X and \c m_X.
    !!
    !! By default, this is done using  by default global \c X_vars variables but
    !! optionally different  limits for the  secondary mode numbers (\c  m_X for
    !! poloidal flux or \c n_X for toroidal flux).
    !!
    !! \note \c n_X and \c m_X need to  have been set up with the same limits as
    !! the grid used here. This is done in setup_nm_X().
    interface set_nm_x
        !> \public
        module procedure set_nm_x_1
        !> \public
        module procedure set_nm_x_2
    end interface

    ! global variables
    type(modes_type) :: mds_x                                          !< modes variables for perturbation grid
    type(modes_type) :: mds_sol                                        !< modes variables for solution grid
    integer :: prim_x                                                  !< \c n_X (pol. flux) or \c m_X (tor. flux)
    integer :: min_sec_x                                               !< \c m_X (pol. flux) or \c n_X (tor. flux) (only for \c X style 1)
    integer :: max_sec_x                                               !< \c m_X (pol. flux) or \c n_X (tor. flux) (only for\ c X style 1)
    integer :: n_mod_x                                                 !< size of \c m_X (pol. flux) or \c n_X (tor. flux)
    integer :: min_nm_x = 5                                            !< minimum for the high-n theory (debable)
    integer, allocatable :: min_n_x(:)                                 !< lowest poloidal mode number \c m_X, in total eq grid
    integer, allocatable :: max_n_x(:)                                 !< highest poloidal mode number \c m_X, in total eq grid
    integer, allocatable :: min_m_x(:)                                 !< lowest poloidal mode number \c m_X, in total eq grid
    integer, allocatable :: max_m_x(:)                                 !< highest poloidal mode number \c m_X, in total eq grid
    real(dp) :: min_r_sol                                              !< min. normal range for pert.
    real(dp) :: max_r_sol                                              !< max. normal range for pert.
#if ldebug
    integer :: n_alloc_x_1s                                            !< nr. of allocated <tt>X_1</tt>'s \ldebug
    integer :: n_alloc_x_2s                                            !< nr. of allocated <tt>X_2</tt>'s \ldebug
#endif

contains
    !> \private vectorial version
    subroutine set_nm_x_1(mds,grid_X,lim_sec_X_o,n_X_loc,m_X_loc,lim_sec_X_i)

        ! input / output
        type(modes_type), intent(in) :: mds                            !< general modes variables
        type(grid_type), intent(in) :: grid_X                          !< perturbation grid
        integer, intent(inout) :: lim_sec_X_o(2)                       !< limits on secondary mode numbers
        integer, intent(inout), allocatable :: n_X_loc(:,:)            !< toroidal mode numbers
        integer, intent(inout), allocatable :: m_X_loc(:,:)            !< poloidal mode numbers
        integer, intent(in), optional :: lim_sec_X_i(2)                !< optional limits on secondary mode numbers

        ! set output lim_sec_X
        lim_sec_x_o = [1,n_mod_x]
        if (present(lim_sec_x_i)) lim_sec_x_o = lim_sec_x_i

        ! set n and m
        allocate(n_x_loc(grid_x%loc_n_r,lim_sec_x_o(2)-lim_sec_x_o(1)+1))
        allocate(m_x_loc(grid_x%loc_n_r,lim_sec_x_o(2)-lim_sec_x_o(1)+1))
        n_x_loc = mds%n(grid_x%i_min:grid_x%i_max,&
            &lim_sec_x_o(1):lim_sec_x_o(2))
        m_x_loc = mds%m(grid_x%i_min:grid_x%i_max,&
            &lim_sec_x_o(1):lim_sec_x_o(2))
    end subroutine set_nm_x_1
    !> \private tensorial version
    subroutine set_nm_x_2(mds,grid_X,lim_sec_X_o,n_X_1,m_X_1,n_X_2,m_X_2,&
        &lim_sec_X_i)

        ! input / output
        type(modes_type), intent(in) :: mds                            !< general modes variables
        type(grid_type), intent(in) :: grid_X                          !< perturbation grid
        integer, intent(inout) :: lim_sec_X_o(2,2)                     !< limits on secondary mode numbers
        integer, intent(inout), allocatable :: n_X_1(:,:)              !< toroidal mode numbers for dimension 1
        integer, intent(inout), allocatable :: m_X_1(:,:)              !< poloidal mode numbers for dimension 1
        integer, intent(inout), allocatable :: n_X_2(:,:)              !< toroidal mode numbers for dimension 2
        integer, intent(inout), allocatable :: m_X_2(:,:)              !< poloidal mode numbers for dimension 2
        integer, intent(in), optional :: lim_sec_X_i(2,2)              !< optional limits on secondary mode numbers

```

```
        ! call vectorial version
        if (present(lim_sec_x_i)) then
            call set_nm_x(mds,grid_x,lim_sec_x_o(:,1),n_x_1,m_x_1,&
                &lim_sec_x_i(:,1))
            call set_nm_x(mds,grid_x,lim_sec_x_o(:,2),n_x_2,m_x_2,&
                &lim_sec_x_i(:,2))
        else
            call set_nm_x(mds,grid_x,lim_sec_x_o(:,1),n_x_1,m_x_1)
            call set_nm_x(mds,grid_x,lim_sec_x_o(:,2),n_x_2,m_x_2)
        end if
    end subroutine set_nm_x_2

    !> \public Initializes a vectorial perturbation.
    !!
    !! Allocates the variables, the number of modes, as well as \c n and \c m .
    !!
    !! Optionally, the secondary mode numbers can be specified (\c m if poloidal
    !! flux is used and \c n if  toroidal flux). By default, they are taken from
    !! the global \c X_vars variables.
    !!
    !! \note If the lowest limits of  the grid is not 1 (e.g. <tt>grid_sol%i_min
    !! = 1</tt> for first process), the input variable \c i_min should be set to
    !! set correctly. For a full grid, it should be set to 1.
    subroutine init_x_1(X,mds,grid_X,lim_sec_X)
#if ldebug
        use num_vars, only: print_mem_usage, rank
#endif

        ! input / output
        class(x_1_type), intent(inout) :: X                                     !< vectorial perturbation variables
        type(modes_type), intent(in) :: mds                                     !< general modes variables
        type(grid_type), intent(in) :: grid_X                                   !< perturbation grid
        integer, intent(in), optional :: lim_sec_X(2)                           !< limits of \c m_X (pol. flux) or \c n_X (tor. flux)

        ! local variables
        integer :: loc_n_r                                                      ! local nr. of normal points
        integer :: n_par, n_geo                                                 ! tot. nr. of angular points in parallel and geodesic direction

        ! set local variables
        loc_n_r = grid_x%loc_n_r
        n_par = grid_x%n(1)
        n_geo = grid_x%n(2)

#if ldebug
        ! initialize memory usage
        if (print_mem_usage) x%estim_mem_usage = 0._dp
#endif

        ! set mode numbers
        call set_nm_x(mds,grid_x,x%lim_sec_X,x%n,x%m,lim_sec_x)

        ! set n_mod
        x%n_mod = size(x%n,2)

        ! allocate U_0
        allocate(x%U_0(n_par,n_geo,loc_n_r,x%n_mod))

        ! allocate U_1
        allocate(x%U_1(n_par,n_geo,loc_n_r,x%n_mod))

        ! allocate DU_0
        allocate(x%DU_0(n_par,n_geo,loc_n_r,x%n_mod))

        ! allocate DU_1
        allocate(x%DU_1(n_par,n_geo,loc_n_r,x%n_mod))

#if ldebug
        ! set estimated memory usage
        if (print_mem_usage) x%estim_mem_usage = x%estim_mem_usage + &
            &(n_par*n_geo*loc_n_r)*(x%n_mod*4)

        ! increment n_alloc_X_1s
        n_alloc_x_1s = n_alloc_x_1s + 1

        ! print memory usage
        if (print_mem_usage) call writo('[rank '//trim(i2str(rank))//&
            &' - Expected memory usage of X_1: '//&
            &trim(r2strt(x%estim_mem_usage*weight_dp*2))//' kB]',alert=.true.)
#endif
    end subroutine init_x_1
    !> \public Initializes a tensorial perturbation.
    !!
    !! \see See init_X_1().
    !!
    !! \note The tensorial perturbation type  can also be used for field-aligned
```

```
    !! variables, in which  case the first index is assumed  to have dimension 1
    !! only.  This can  be triggered  using  \c is_field_averaged.  There is  no
    !! difference  between a  tensorial  perturbation type  with  size of  first
    !! dimension  set to  one through  the use  of this  flag, or  through other
    !! means.
    subroutine init_x_2(X,mds,grid_X,lim_sec_X,is_field_averaged)
#if ldebug
        use num_vars, only: print_mem_usage, rank
#endif

        ! input / output
        class(x_2_type), intent(inout) :: X                                     !< tensorial perturbation variables
        type(modes_type), intent(in) :: mds                                     !< general modes variables
        type(grid_type), intent(in) :: grid_X                                   !< perturbation grid
        integer, intent(in), optional :: lim_sec_X(2,2)                         !< limits of \c m_X (pol. flux) or \c n_X (tor. flux) for both dimensions
        logical, intent(in), optional :: is_field_averaged                      !< if field-aligned, only one dimension for first index

        ! local variables
        integer :: loc_n_r                                                      ! local nr. of normal points
        integer :: n_par, n_geo                                                 ! tot. nr. of angular points in parallel and geodesic direction
        integer :: nn_mod                                                       ! either n_mod^2, n_mod*(n_mod+1)/2 or 0

        ! set local variables
        n_par = grid_x%n(1)
        if (present(is_field_averaged)) then
            if (is_field_averaged) n_par = 1
        end if
        n_geo = grid_x%n(2)
        loc_n_r = grid_x%loc_n_r

#if ldebug
        ! initialize memory usage
        if (print_mem_usage) x%estim_mem_usage = 0._dp
#endif

        ! set mode numbers
        call set_nm_x(mds,grid_x,x%lim_sec_X,x%n_1,x%m_1,x%n_2,x%m_2,lim_sec_x)

        ! set n_mod
        x%n_mod(1) = size(x%n_1,2)
        x%n_mod(2) = size(x%n_2,2)

        ! set nnmod for symmetric quantities
        nn_mod = set_nn_mod(.true.,lim_sec_x)

        ! allocate PV_i
        allocate(x%PV_0(n_par,n_geo,loc_n_r,nn_mod))                            ! symmetric
        allocate(x%PV_1(n_par,n_geo,loc_n_r,product(x%n_mod)))                  ! not symmetric
        allocate(x%PV_2(n_par,n_geo,loc_n_r,nn_mod))                            ! symmetric

        ! allocate KV_i
        allocate(x%KV_0(n_par,n_geo,loc_n_r,nn_mod))                            ! symmetric
        allocate(x%KV_1(n_par,n_geo,loc_n_r,product(x%n_mod)))                  ! not symmetric
        allocate(x%KV_2(n_par,n_geo,loc_n_r,nn_mod))                            ! symmetric

#if ldebug
        ! set estimated memory usage
        if (print_mem_usage) x%estim_mem_usage = &
            &x%estim_mem_usage + (n_par*n_geo*loc_n_r)*&
            &(nn_mod*4+product(x%n_mod)*2) + product(x%n_mod)

        ! increment n_alloc_X_2s
        n_alloc_x_2s = n_alloc_x_2s + 1

        ! print memory usage
        if (print_mem_usage) call writo('[rank '//trim(i2str(rank))//&
            &' - Expected memory usage of X_2: '//&
            &trim(r2strt(x%estim_mem_usage*weight_dp*2))//' kB]',alert=.true.)
#endif
    end subroutine init_x_2

    !> \public Deep copy of vectorial perturbation variables.
    subroutine copy_x_1(X_i,mds,grid_i,X_o)
        use grid_vars, only: grid_type

        ! input / output
        class(x_1_type), intent(in) :: X_i                                      !< X_1 to be copied
        type(modes_type), intent(in) :: mds                                     !< general modes variables
        type(grid_type), intent(in) :: grid_i                                   !< grid of eq_i
        type(x_1_type), intent(inout) :: X_o                                    !< copied X_1

        call x_o%init(mds,grid_i,lim_sec_x=x_i%lim_sec_X)

        ! U_i
```

```
        x_o%U_0 = x_i%U_0
        x_o%U_1 = x_i%U_1

        ! DU_i
        x_o%DU_0 = x_i%DU_0
        x_o%DU_1 = x_i%DU_1
    end subroutine copy_x_1

    !> \public Deep copy of tensorial perturbation variables.
    subroutine copy_x_2(X_i,mds,grid_i,X_o)
        use grid_vars, only: grid_type

        ! input / output
        class(x_2_type), intent(in) :: X_i                                  !< X_2 to be copied
        type(modes_type), intent(in) :: mds                                 !< general modes variables
        type(grid_type), intent(in) :: grid_i                               !< grid of eq_i
        type(x_2_type), intent(inout) :: X_o                                !< copied X_1

        call x_o%init(mds,grid_i,lim_sec_x=x_i%lim_sec_X,&
            &is_field_averaged=size(x_i%PV_0,1).eq.1)

        ! PV_i
        x_o%PV_0 = x_i%PV_0
        x_o%PV_1 = x_i%PV_1
        x_o%PV_2 = x_i%PV_2

        ! KV_i
        x_o%KV_0 = x_i%KV_0
        x_o%KV_1 = x_i%KV_1
        x_o%KV_2 = x_i%KV_2
    end subroutine copy_x_2

    !> Sets number of entries for tensorial perturbation variables.
    integer function set_nn_mod(sym,lim_sec_X) result(nn_mod)
        ! input / output
        logical, intent(in) :: sym                                          !< whether the variable is symmetric
        integer, intent(in), optional :: lim_sec_x(2,2)                     !< limits of \c m_X (pol flux) or \c n_X
 (tor flux) for both dimensions

        ! local variables
        integer :: id, jd                                                   ! counters
        integer :: lim_sec_x_loc(2,2)                                       ! local version of lim_sec_X

        ! set local lim_sec_X
        lim_sec_x_loc(:,1) = [1,n_mod_x]
        lim_sec_x_loc(:,2) = [1,n_mod_x]
        if (present(lim_sec_x)) lim_sec_x_loc = lim_sec_x

        if (sym) then
            ! set nnmod for symmetric quantities: discard indices above diagonal
            nn_mod = 0
            do jd = lim_sec_x_loc(1,2),lim_sec_x_loc(2,2)
                do id = lim_sec_x_loc(1,1),lim_sec_x_loc(2,1)
                    if (id.ge.jd) nn_mod = nn_mod + 1
                end do
            end do
        else
            ! set nn_mod for asymmetric quantities: don't discard anything
            nn_mod = product(lim_sec_x_loc(2,:)-lim_sec_x_loc(1,:)+1)
        end if
    end function set_nn_mod

    !> \public Deallocates modes variables
    subroutine dealloc_mds(mds)
        ! input / output
        class(modes_type), intent(inout) :: mds                             !< modes variables to be deallocated

        ! deallocate allocatable variables
        call dealloc_mds_final(mds)
    contains
        ! Note: intent(out) automatically deallocates the variable
        !> \private
        subroutine dealloc_mds_final(mds)
            ! input / output
            type(modes_type), intent(out) :: mds                            ! modes variables to be deallocated
        end subroutine dealloc_mds_final
    end subroutine dealloc_mds

    !> \public Deallocates vectorial perturbation variables.
    subroutine dealloc_x_1(X)
#if ldebug
        use num_vars, only: rank, print_mem_usage
#endif

        ! input / output
        class(x_1_type), intent(inout) :: X                                 !< perturbation variables to be deallocated

```

```
#if ldebug
        ! local variables
        integer :: mem_diff                                             ! difference in memory
        real(dp) :: estim_mem_usage                                     ! estimated memory usage

        ! memory usage before deallocation
        if (print_mem_usage) then
            mem_diff = get_mem_usage()
            estim_mem_usage = x%estim_mem_usage
        end if
#endif

        ! deallocate allocatable variables
        call dealloc_x_1_final(x)

#if ldebug
        ! decrement n_alloc_X_1s
        n_alloc_x_1s = n_alloc_x_1s - 1

        ! memory usage difference after deallocation
        if (print_mem_usage) then
            mem_diff = mem_diff - get_mem_usage()
            call writo('[Rank '//trim(i2str(rank))//' - liberated '//&
                &trim(i2str(mem_diff))//'kB deallocating X_1 ('//&
                &trim(i2str(nint(100*mem_diff/&
                &(estim_mem_usage*weight_dp*2)))))//&
                &'% of estimated)]',alert=.true.)
        end if
#endif
    contains
        ! Note: intent(out) automatically deallocates the variable
        !> \private
        subroutine dealloc_x_1_final(X)
            ! input / output
            type(x_1_type), intent(out) :: X                            ! equilibrium to be deallocated
        end subroutine dealloc_x_1_final
    end subroutine dealloc_x_1
    !> \public Deallocates tensorial perturbation variables.
    subroutine dealloc_x_2(X)
#if ldebug
        use num_vars, only: rank, print_mem_usage
#endif

        ! input / output
        class(x_2_type), intent(inout) :: X                             !< perturbation variables to be deallocated

#if ldebug
        ! local variables
        integer :: mem_diff                                             ! difference in memory
        real(dp) :: estim_mem_usage                                     ! estimated memory usage

        ! memory usage before deallocation
        if (print_mem_usage) then
            mem_diff = get_mem_usage()
            estim_mem_usage = x%estim_mem_usage
        end if
#endif

        ! deallocate allocatable variables
        call dealloc_x_2_final(x)

#if ldebug
        ! decrement n_alloc_X_2s
        n_alloc_x_2s = n_alloc_x_2s - 1

        ! memory usage difference after deallocation
        if (print_mem_usage) then
            mem_diff = mem_diff - get_mem_usage()
            call writo('[Rank '//trim(i2str(rank))//' - liberated '//&
                &trim(i2str(mem_diff))//'kB deallocating X_2 ('//&
                &trim(i2str(nint(100*mem_diff/&
                &(estim_mem_usage*weight_dp*2)))))//&
                &'% of estimated)]',alert=.true.)
        end if
#endif
    contains
        ! Note: intent(out) automatically deallocates the variable
        !> \private
        subroutine dealloc_x_2_final(X)
            ! input / output
            type(x_2_type), intent(out) :: X                            ! equilibrium to be deallocated
        end subroutine dealloc_x_2_final
    end subroutine dealloc_x_2
end module x_vars
```

D.104 PB3D.f90

Go to the documentation of this file.

```
! (from http://patorjk.com/software/taag/ ANSI shadow)

!   ▄▄▄▄▄▄▄▄ ▄▄▄▄▄▄▄▄ ▄▄▄▄▄▄▄▄ ▄▄▄▄▄▄▄▄
!   ▐▐▐▐▐▐▐▐ ▐▐▐▐▐▐▐▐ ▐▐▐▐▐▐▐▐ ▐▐▐▐▐▐▐▐
!   ▐▐▐▐▐▐▐▌ ▐▐▐▐▐▐▐▌ ▐▐▐▐▐▐▐▌ ▐▐▐▐▐▐▐▌
!   ▐▐▐▌    ▐▐▐▐▐▐▐▐ ▐▐▐▌▐▐▐▐ ▐▐▐▌▐▐▐▐
!   ▐▐▐▌    ▐▐▐▐▐▐▐▌ ▐▐▐▐▐▐▐▌ ▐▐▐▐▐▐▐▌
!   ▐▐▐▌    ▐▐▐▐▐▐▐▐ ▐▐▐▐▐▐▐▐ ▐▐▐▐▐▐▐▐

!-----------------------------------------------------------------------------!
!>  Peeling Ballooning in 3D
!-----------------------------------------------------------------------------!
!>  \author
!!  Toon Weyens,
!!  Contact: weyenst@gmail.com
!-----------------------------------------------------------------------------!
!>  \version    2.41
!!  \date       2012-2020
!!  Copyright (C) 2020Toon Weyens
!!
!!  This program is free software: you can redistribute it and/or modify
!!  it under the terms of the GNU General Public License as published by
!!  the Free Software Foundation, either version 3 of the License, or
!!  (at your option) any later version.
!!
!!  This program is distributed in the hope that it will be useful,
!!  but WITHOUT ANY WARRANTY; without even the implied warranty of
!!  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
!!  GNU General Public License for more details.
!!
!!  You should have received a copy of the License along with this program.
!!  If not, contact weyenst@gmail.com
!-----------------------------------------------------------------------------!
!>  \see
!!  References:
!!  \cite weyens2014theory
!!  \cite Weyens2017PB3D
!-----------------------------------------------------------------------------!
#define CHCKERR if(ierr.ne.0) then; call sudden_stop(ierr); end if
program pb3d
    use num_vars, only: prog_name, prog_style, rank, rich_restart_lvl, eq_style
    use str_utilities, only: r2str, i2str
    use messages
    use grid_vars, only: grid_type
    use eq_vars, only: eq_1_type, eq_2_type
    use x_vars, only: x_1_type, x_2_type
    use vac_vars, only: vac_type
    use sol_vars, only: sol_type
    use hdf5_vars, only: init_hdf5
    use driver_eq, only: run_driver_eq
    use driver_x, only: run_driver_x
    use driver_sol, only: run_driver_sol
    use files_ops, only: open_input, open_output, parse_args, init_files, &
        &close_output
    use input_ops, only: read_input_opts, read_input_eq, print_output_in
    use input_utilities, only: dealloc_in
    use mpi_ops, only: start_mpi, stop_mpi, broadcast_input_opts, sudden_stop
    use eq_ops, only: calc_normalization_const, normalize_input
    use eq_utilities, only: do_eq, eq_info
    use rich_ops, only: init_rich, term_rich, do_rich, start_rich_lvl, &
        &stop_rich_lvl
    use rich_vars, only: rich_info, &
        &rich_lvl
#if ldebug
    use num_vars, only: ltest
    use test, only: generic_tests
#endif

    implicit none

    ! local variables
    integer :: ierr                                                     ! error
    type(grid_type), target :: grid_eq                                  ! equilibrium grid
    type(grid_type), pointer :: grid_eq_b => null()                     ! field-aligned equilibrium grid
    type(grid_type), target :: grid_x                                   ! perturbation grid
    type(grid_type), pointer :: grid_x_b => null()                      ! field-aligned perturbation grid
    type(grid_type) :: grid_sol                                         ! solution grid
    type(eq_1_type) :: eq_1                                             ! flux equilibrium variables
    type(eq_2_type) :: eq_2                                             ! metric equilibrium variables
    type(x_1_type) :: x_1                                               ! vectorial perturbation variables
    type(x_2_type) :: x_2                                               ! integrated tensorial perturbation
variables
```

```
    type(vac_type) :: vac                                                   ! vacuum variables
    type(sol_type) :: sol                                                   ! solution variables

    !-----------------------------!
    !   Initialize some routines   !
    !-----------------------------!
    ierr = start_mpi()                                                      ! start MPI
    chckerr
    prog_name = 'PB3D'                                                      ! program name
    prog_style = 1                                                          ! main part
    call print_hello()                                                      ! print message with time, etc
    call init_output()                                                      ! initialize output utilities
    call init_files()                                                       ! initialize file operations
    call init_time()                                                        ! initialize time
    call init_hdf5()                                                        ! initialize HDF5

    !-------------------------------------------------------------------!
    !   Read the user-provided input file and the equilibrium code output   !
    !-------------------------------------------------------------------!
    call start_time
    call writo('Initialization')
    call lvl_ud(1)
    if (rank.eq.0) then                                                     ! only master
        ierr = parse_args()                                                 ! parse argument (options are used in open_input)
        chckerr
        ierr = open_input()                                                 ! open the input files
        chckerr
        ierr = read_input_opts()                                            ! read input options file
        chckerr
        if (rich_restart_lvl.eq.1) then                                     ! start at first level
            ierr = read_input_eq()                                          ! read input equilibrium file
            chckerr
            ierr = calc_normalization_const()                               ! set up normalization constants
            chckerr
            call normalize_input()                                          ! normalize the input
            ierr = open_output()                                            ! open output file
            chckerr
            ierr = print_output_in('in')                                    ! print input outputs
            chckerr
            call dealloc_in()                                               ! clean up input from equilibrium codes
        else                                                                ! restart at higher level
            ierr = open_output()                                            ! open output file
            chckerr
        end if
    end if
    ierr = broadcast_input_opts()                                           ! broadcast input options to other processors
    chckerr
    call writo('')
    call passed_time
    call writo('')
    call lvl_ud(-1)

#if ldebug
    !-------------------!
    !   Do some tests   !
    !-------------------!
    if (ltest) then
        call start_time
        call writo('Generic Tests')
        call lvl_ud(1)
        ierr = generic_tests()                                              ! run generic test
        chckerr
        call writo('')
        call passed_time
        call writo('')
        call lvl_ud(-1)
    end if
#endif

    !---------------------------------------------!
    !   Initialize Richardson Extrapolation Loop   !
    !---------------------------------------------!
    call start_time
    ierr = init_rich()
    chckerr
    call stop_time

    rich: do while(do_rich())
        !---------------------------!
        !   Start Richardson level   !
        !---------------------------!
        call start_time
        call writo('Richardson level '//trim(i2str(rich_lvl)))
        call lvl_ud(1)
        ierr = start_rich_lvl()                                             ! start Richardson level, setting n_r_sol
```

```
    and other variables
        chckerr
        call stop_time
        call writo('')
        call lvl_ud(-1)

        par: do while(do_eq())
            !-----------------------------------!
            !   Main Driver: Equilibrium part   !
            !-----------------------------------!
            call start_time
            call writo('Equilibrium driver'//trim(rich_info())//&
                &trim(eq_info()))
            call lvl_ud(1)
            ierr = run_driver_eq(grid_eq,grid_eq_b,eq_1,eq_2,vac)               ! equilibrium driver
            chckerr
            call writo('')
            call passed_time
            call writo('')
            call lvl_ud(-1)

            !-----------------------------------!
            !   Main driver: Perturbation part   !
            !-----------------------------------!
            call start_time
            call writo('Perturbation driver'//trim(rich_info())//&
                &trim(eq_info()))
            call lvl_ud(1)
            ierr = run_driver_x(grid_eq,grid_eq_b,grid_x,grid_x_b,eq_1,eq_2,&
                &x_1,x_2)                                                       ! perturbation driver
            chckerr
            call writo('')
            call passed_time
            call writo('')
            call lvl_ud(-1)
        end do par

        !-------------------------------!
        !   Main driver: Solution part   !
        !-------------------------------!
        call start_time
        call writo('Solution driver'//trim(rich_info()))
        call lvl_ud(1)
        ierr = run_driver_sol(grid_eq,grid_x,grid_sol,x_2,vac,sol)              ! solution driver
        chckerr
        call writo('')
        call passed_time
        call writo('')
        call lvl_ud(-1)

        call start_time
        call stop_rich_lvl()
        call stop_time
    end do rich

    !--------------------------------------!
    !   Stop Richarson Extrapolation Loop   !
    !--------------------------------------!
    call start_time
    call term_rich()                                                            ! stop Richardson loop
    call stop_time

    !--------------!
    !   Clean up   !
    !--------------!
    call writo('Clean up')
    call lvl_ud(1)
    if (eq_style.eq.2) call eq_2%dealloc()
    ierr = stop_mpi(grid_eq=grid_eq,&
        &grid_eq_b=grid_eq_b,&
        &grid_x=grid_x,&
        &grid_x_b=grid_x_b,&
        &grid_sol=grid_sol,&
        &eq_1=eq_1,&
        &eq_2=eq_2,&
        &x_1=x_1,&
        &x_2=x_2,&
        &vac=vac,&
        &sol=sol)
    chckerr
    call close_output
    call lvl_ud(-1)

    call print_goodbye
end program pb3d
```

D.105 POST.f90

Go to the documentation of this file.

```
! (from http://patorjk.com/software/taag/ ANSI shadow)

!   ▄▄▄▄▄▄▄ ▄▄▄▄▄▄▄ ▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄
!   ▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐
!   ▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐
!   ▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐
!   ▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐
!   ▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐
!   ▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐

!-----------------------------------------------------------------------!
!>  Peeling Ballooning in 3D: postprocessing
!-----------------------------------------------------------------------!
!>  \author
!!  Toon Weyens,
!!  Contact: weyenst@gmail.com
!-----------------------------------------------------------------------!
!>  \version    2.41
!!  \date       2012-2018
!!  Copyright (C) 2019 Toon Weyens
!!
!!  This program is free software: you can redistribute it and/or modify
!!  it under the terms of the GNU General Public License as published by
!!  the Free Software Foundation, either version 3 of the License, or
!!  (at your option) any later version.
!!
!!  This program is distributed in the hope that it will be useful,
!!  but WITHOUT ANY WARRANTY; without even the implied warranty of
!!  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
!!  GNU General Public License for more details.
!!
!!  You should have received a copy of the License along with this program.
!!  If not, contact weyenst@gmail.com
!-----------------------------------------------------------------------!
!>  \see
!!  References:
!!  \cite weyens2014theory
!!  \cite Weyens2017PB3D
!-----------------------------------------------------------------------!
#define CHCKERR if(ierr.ne.0) then; call sudden_stop(ierr); end if
program post
    use str_utilities, only: i2str
    use num_vars, only: prog_name, prog_style, rank
    use messages
    use hdf5_vars, only: init_hdf5
    use mpi_ops, only: start_mpi, stop_mpi, broadcast_input_opts, &
        &sudden_stop
    use files_ops, only: init_files, parse_args, open_input, open_output, &
        &close_output
    use input_ops, only: read_input_opts
    use driver_post, only: run_driver_post, init_post, stop_post
    use pb3d_ops, only: reconstruct_pb3d_in
    use input_utilities, only: dealloc_in
    use eq_utilities, only: do_eq, eq_info
    use grid_vars, only: grid_type
#if ldebug
    use num_vars, only: ltest
    use test, only: generic_tests
#endif

    implicit none

    ! local variables
    integer :: ierr                                                     ! error

    !-----------------------------!
    !   Initialize some routines   !
    !-----------------------------!
    ierr = start_mpi()                                                  ! start MPI
    chckerr
    prog_name = 'POST'                                                  ! program name
    prog_style = 2                                                      ! post-processing part
    call print_hello()                                                  ! print message with time, etc
    call init_output()                                                  ! initialize output
    call init_files()                                                   ! initialize file operations
    call init_time()                                                    ! initialize time
    call init_hdf5()                                                    ! initialize HDF5

    !-------------------------!
    !   Read the PB3D output   !
    !-------------------------!
    call start_time
```

```
    call writo('Initialization')
    call lvl_ud(1)
    if (rank.eq.0) then
        ierr = parse_args()                                         ! parse argument (options are used in open_input)
        chckerr
        ierr = open_input()                                         ! open the input files
        chckerr
        ierr = reconstruct_pb3d_in('in')                            ! reconstruct miscellaneous PB3D output variables
        chckerr
        ierr = read_input_opts()                                    ! read input options file
        chckerr
        ierr = open_output()                                        ! open output file per alpha group
        chckerr
        call dealloc_in()                                           ! clean up input from equilibrium codes
    end if
    ierr = broadcast_input_opts()                                   ! broadcast input options to other processors
    chckerr
    call writo('')
    call passed_time
    call writo('')
    call lvl_ud(-1)

#if ldebug
    !-------------------!
    !   Do some tests   !
    !-------------------!
    if (ltest) then
        call start_time
        call writo('Generic Tests')
        call lvl_ud(1)
        ierr = generic_tests()
        chckerr
        call writo('')
        call passed_time
        call writo('')
        call lvl_ud(-1)
    end if
#endif

    !-----------------!
    !   Main driver   !
    !-----------------!
    call start_time
    call writo('Initializing POST')
    call lvl_ud(1)
    ierr = init_post()
    chckerr
    call stop_time
    call writo('')
    call lvl_ud(-1)
    par: do while(do_eq())
        call start_time
        call writo('Main driver'//trim(eq_info()))
        call lvl_ud(1)
        ierr = run_driver_post()
        chckerr
        call writo('')
        call passed_time
        call writo('')
        call lvl_ud(-1)
    end do par
    call stop_post()

    !--------------!
    !   clean up   !
    !--------------!
    call writo('Clean up')
    call lvl_ud(1)
    ierr = stop_mpi()
    chckerr
    call close_output
    call lvl_ud(-1)

    call print_goodbye
end program post
```

Appendix E

Examples

E.1 PB3D.input

Example PB3D input.

```
&inputdata_PB3D
    min_n_par_X             = 500
    min_par_X               = 0.0
    max_par_X               = 2.0
    alpha                   = 0.0
    min_r_sol               = 0.1
    max_r_sol               = 1.0
    n_r_sol                 = 500
    prim_X                  = 10
    n_mod_X                 = 20
    BC_style                = 1 1
    U_style                 = 3
    max_tot_mem             = 8000
    max_it_rich             = 2
    use_normalization       = .true.
/
//! [some extra options for PB3D]
    rich_restart_lvl        = 1
    tol_rich                = 1.0E-4
    tol_SLEPC               = 1.0E-8 1.0E-8
//! [some extra options for PB3D]
```

E.2 POST.input

Example POST input.

```
&INPUTDATA_POST
    plot_resonance          = .true.
    plot_magn_grid          = .true.
    plot_flux_q             = .true.
    plot_sol_xi             = .true.
    plot_sol_Q              = .true.
    plot_E_rec              = .true.
    plot_B                  = .true.
    plot_J                  = .true.
    plot_kappa              = .true.
    min_r_plot              = 0.0
    max_r_plot              = 1.0
    n_theta_plot            = 101
    min_theta_plot          = 0.00
    max_theta_plot          = 2.00
    n_zeta_plot             = 1
    min_zeta_plot           = 0.00
    max_zeta_plot           = 0.00
    POST_style              = 1
    plot_grid_style         = 0
    pert_mult_factor_POST   = 0.0000
    max_tot_mem             = 8000
/
//! [some extra options for POST]
    ex_plot_style           = 2
    PB3D_rich_lvl           = 1
//! [some extra options for POST]
```

E.3 example.mk

An example makefile:

```
################################################################################
#
#   Example makefile for the program PB3D (Peeling Ballooning in 3D)
#   \author Author: Toon Weyens
#
#   Don't forget to set the directories:
#       - LIBSTELL_DIR
#       - HDF5_DIR
#       - NETCDFF_DIR (note: Fortran library)
#       - PETSC_DIR
#       - SLEPC_DIR
#       - STRUMPACK_DIR
################################################################################

################################################################################
#   Include
################################################################################
## [PETSc and SLEPc trick]
include  $(PETSC_DIR)/lib/petsc/conf/variables
include  $(SLEPC_DIR)/lib/slepc/conf/slepc_variables
## [PETSc and SLEPc trick]

## [PETSc and SLEPc trick inc]
INCLUDE = $(PETSC_FC_INCLUDES) $(SLEPC_INCLUDE)
## [PETSc and SLEPc trick inc]
## [Libstell special]
INCLUDE += -I$(LIBSTELL_DIR)/libstell_dir
## [Libstell special]
INCLUDE += -I$(STRUMPACK_DIR)/include
## [PB3D include]
INCLUDE += -I$(PB3D_DIR)/include
## [PB3D include]
INCLUDE += -I/usr/include/hdf5/openmpi

################################################################################
#   Link
################################################################################
## [PB3D libraries]
LIB_INTERNAL = libdfftpack.a libfoul.a libbspline.a
## [PB3D libraries]

LINK := $(LIB_INTERNAL)

## [PETSc and SLEPc trick lib]
LINK += $(PETSC_LIB)
LINK += $(SLEPC_LIB)
## [PETSc and SLEPc trick lib]
LINK += $(LIBSTELL_DIR)/libstell.a
LINK += -L$(STRUMPACK_DIR)/lib -lstrumpack
LINK += -L$(HDF5_DIR) -lhdf5_fortran -lhdf5
LINK += -L$(NETCDFF_DIR)/lib -lnetcdff
LINK += -Wl,-R$(NETCDFF_DIR)/lib
LINK += -lscalapack -lblacs -lblas -lm
LINK += -lstdc++ -lmpi_cxx


################################################################################
#   Compiler
################################################################################
COMPILER=mpifort


################################################################################
#   Linker
################################################################################
LINKER=mpifort


################################################################################
#   Compiler flags
#   options (used with -D[name]):
#       ldebug: debug
#       lIB: infiniband
#       lwith_gnu: use GNU compiler [default]
#       lwith_intel: use INTEL compiler, (checked for version 12.0.2)
#   note: INTEL warning 6536 is suppressed, which informs about extra "USE".
#   note: INTEL warning 6843 is suppressed, which informs about empty
#       intent(out) variables
################################################################################
COMP_FLAGS = -finit-real=snan -g -Og -Wall -Wextra -pedantic \
    -fimplicit-none -fbacktrace -fno-omit-frame-pointer \
```

```
    -fcheck=all -cpp -Dldebug# debug, profiling with gprof2dot, GCC
#COMP_FLAGS = -O3 -fbacktrace -g -fimplicit-none -fno-omit-frame-pointer \
    #-cpp# optimized, GCC

#COMP_FLAGS = -O0 -DlIB -Dldebug -g -heap-arrays 100 -recursive \
    #-ftrapuv -check bounds -check uninit -traceback -implicitnone \
    #-fno-omit-frame-pointer -cpp -Dlwith_intel -diag-disable 6536 \
    #-diag-disable 6843# debug, profiling with gprof2dot, INTEL
#COMP_FLAGS = -O3 -DlIB -traceback -g -heap-arrays 100 -recursive \
    #-implicitnone -fno-omit-frame-pointer -cpp -Dlwith_intel \
    #-diag-disable 6536 -diag-disable 6843# optimized, INTEL

COMP_FLAGS_EX= -O2 -w

COMP_FLAGS_F= -O2 -funroll-loops -fexpensive-optimizations


################################################################################
#   Link flags
################################################################################
LINK_FLAGS = -fPIC -finit-real=snan# debug
#LINK_FLAGS = -fPIC# optimized


################################################################################
#   Prepare
################################################################################
# Add "Modules" and "Libraries" to the search path for the prerequisites
VPATH = Modules:Libraries

# Contains list of source files (.o) and dependencies
DEPLIST = PB3D.dep
OBJLIST = ObjectList# defines "ObjectFiles"

# Includes source files and dependency list
include $(DEPLIST)# Dependencies of all the objects
include $(OBJLIST)# Names of all the objects


################################################################################
#   Rules
################################################################################
all:    PB3D POST

PB3D:   $(ObjectFiles) $(LIB_INTERNAL) PB3D.o
    $(LINKER) -o $@ $(ObjectFiles) PB3D.o $(LINK) $(LINK_FLAGS)

POST:   $(ObjectFiles) $(LIB_INTERNAL) POST.o
    $(LINKER) -o $@ $(ObjectFiles) POST.o $(LINK) $(LINK_FLAGS)

libdfftpack.a:  dfft.o
    ar -rcs libdfftpack.a dfft.o

libfoul.a:  foul.o
    ar -rcs libfoul.a foul.o

libbspline.a:   bspline_sub_module.o
    ar -rcs libbspline.a bspline_sub_module.o

%.o: %.f90
    $(COMPILER) $(INCLUDE) $(COMP_FLAGS) -c $<

%.o: %.f
    $(COMPILER) $(COMP_FLAGS_F) -c $<

dfft.o: dfft.f
    $(COMPILER) $(COMP_FLAGS_EX) -c $<

foul.o: foul.f90
    $(COMPILER) $(COMP_FLAGS_EX) -c $<

bspline_sub_module.o: bspline_sub_module.f90
    $(COMPILER) $(COMP_FLAGS_EX) -c $<

clean:
    @rm -f *.o *.a *.mod *~ fort.*

clean_all:
    @rm -f *.o *.mod *~ fort.* PB3D POST
```

Bibliography

[1] Sivaram Ambikasaran. Fast Algorithms for Dense Numerical Linear Algebra and Applications. (August), 2013. 48

[2] Ake Bjorck and Victor Pereyra. Solutions of Vandermonde Systems of Equations. *Mathematics of Computation*, 24(12):893–903, 1970. 247

[3] Howard S. Cohl and Joel E. Tohline. A Compact Cylindrical Green's Function Expansion for the Solution of Potential Problems. *The Astrophysical Journal*, 527(1):86–101, dec 1999. 350

[4] G. Dahlquist and Å. Björk. *Numerical Methods*. Dover Publications, dover book edition, 2013. 298

[5] N. M. Ferraro, S. C. Jardin, and P. B. Snyder. Ideal and resistive edge stability calculations with M3D-C1. *Physics of Plasmas*, 17(10):102508, oct 2010. 43

[6] Per Helander. Theory of plasma confinement in non-axisymmetric magnetic fields. *Reports on Progress in Physics*, 77(8):087001, aug 2014. 350, 383

[7] Vicente Hernandez, Jose E. Roman, and Vicente Vidal. SLEPc. *ACM Transactions on Mathematical Software*, 31(3):351–362, sep 2005. 308, 319

[8] S. P. Hirshman. Steepest-descent moment method for three-dimensional magnetohydrodynamic equilibria. *Physics of Fluids*, 26(12):3553, 1983. 39

[9] P. Holoborodko. Smooth Noise Robust Differentiators, 2008. 232, 238, 928

[10] Dominic Meiser. Replicated Computational Results (RCR) Report for "A Distributed-Memory Package for Dense Hierarchically Semi-Separable Matrix Computations Using Randomization". *ACM Transactions on Mathematical Software*, 42(4):1–5, jul 2016. 350

[11] A. B. Mikhailovskii, G.T.A. Huysmans, W. O. K. Kerner, and S. E. Sharapov. Optimisation of Computational MHD Normal-mode Analysis in Tokamaks. *Plasma Physics Reports*, 23(10):844–857, 1997. 39

[12] Cooper Redwine. Upgrading to FORTRAN 90. *Book*, page 514, 1995. 347

[13] Robert Ross, Robert Latham, William Gropp, Rajeev Thakur, Brian Toonen, and Computer Science Division. Implementing MPI-IO Atomic Mode Without File System Support. 220, 222, 223, 224

[14] J. Segura and A. Gil. Evaluation of toroidal harmonics. *Computer Physics Communications*, 124(1):104–122, jan 2000. 79

[15] T. Weyens. Three-dimensional linear peeling-ballooning theory in magnetic fusion devices. page 200. 84, 85, 86, 100, 182, 183, 187, 323, 350, 428, 438, 449, 451, 453, 459, 490, 546, 710, 842

[16] T. Weyens, R. Sánchez, L. García, A. Loarte, and G. Huijsmans. Three-dimensional linear peeling-ballooning theory in magnetic fusion devices. *Physics of Plasmas*, 21(4):042507, apr 2014. 34, 60, 62

[17] T. Weyens, R. Sánchez, G. Huijsmans, A. Loarte, and L. García. PB3D: A new code for edge 3-D ideal linear peeling-ballooning stability. *Journal of Computational Physics*, 330:997–1009, feb 2017. 1, 33, 43, 60, 62, 313

[18] XDMF. XDMF Model and Format, 2015. 514

[19] Z. Zhang. An improvement to the Brent's method. *International Journal of Experimental Algorithms*, 2(1):21–26, 2011. 230