MAGIC Documentation

Release 5.3

The MAGIC dev team

December 03, 2015

1 Introduction **1**
1.1 Foreword . . . 1
1.2 Giving credit . . . 2

2 Get MagIC and run it **3**
2.1 Download the code . . . 3
2.2 Setting up the environment variables . . . 3
2.3 Setting up compiler options and compiling . . . 4
2.4 Preparing a production run . . . 6

3 Formulation of the (magneto)-hydrodynamics problem **7**
3.1 The reference state . . . 9
3.2 Boussinesq approximation . . . 10
3.3 Anelastic approximation . . . 11
3.4 Dimensionless control parameters . . . 13
3.5 Boundary conditions and treatment of inner core . . . 14

4 Numerical technique **17**
4.1 Poloidal/toroidal decomposition . . . 17
4.2 Spherical harmonic representation . . . 18
4.3 Radial representation . . . 21
4.4 Spectral equations . . . 22
4.5 Time-stepping schemes . . . 26
4.6 Coriolis force and nonlinear terms . . . 28
4.7 Boundary conditions and inner core . . . 33

5 Contributing to the code **37**
5.1 Checking the consistency of the code . . . 37
5.2 Advices when contributing to the code . . . 38
5.3 Building the documentation and contributing to it . . . 39

6 Input parameters **41**
6.1 Grid namelist . . . 42
6.2 Control namelist . . . 43
6.3 Physical parameters namelist . . . 46
6.4 External Magnetic Field Namelist . . . 52
6.5 Start field namelist . . . 53
6.6 Output control namelist . . . 56
6.7 Mantle and Inner Core Namelists . . . 68

7 Interactive communication with the code using `signal.TAG` **71**

8 Output files **73**
8.1 Log file: `log.TAG` . . . 74
8.2 Default time-series outputs . . . 75
8.3 Additional optional time-series outputs . . . 80
8.4 Time-averaged radial profiles . . . 85
8.5 Transport properties of the reference state . . . 90
8.6 Nonlinear mapping of the Chebyshev grid . . . 91
8.7 Spectra . . . 92
8.8 Graphic files `G_#.TAG` and `G_ave.TAG` . . . 96
8.9 Movie files `*_mov.TAG` . . . 101
8.10 Restart files `rst_*.TAG` . . . 104
8.11 Poloidal and toroidal potentials at given depths . . . 106
8.12 `TO` outputs . . . 111
8.13 Radial spectra `rB[r|p]Spec.TAG` . . . 112
8.14 Potential files `[V|B|T]pot_#.TAG` . . . 113

9 Data visualisation and post-processing **117**
9.1 Requirements . . . 117
9.2 Configuration: `magic.cfg` file . . . 117
9.3 Python functions and classes . . . 119

10 Description of the Fortran modules **151**
10.1 Main program `magic.f90` . . . 151
10.2 Base modules . . . 152
10.3 MPI related modules . . . 179
10.4 Code initialization . . . 186
10.5 Pre-calculations . . . 193
10.6 Time stepping . . . 201
10.7 Linear calculation part of the time stepping (LMLoop) . . . 208
10.8 Non-linear part of the time stepping (radial loop) . . . 220
10.9 Chebyshev polynomials and cosine transforms . . . 234
10.10 Legendre transforms . . . 241
10.11 Fourier transforms . . . 252
10.12 Linear algebra . . . 257
10.13 Radial derivatives and integration . . . 259
10.14 Blocking and LM mapping . . . 268
10.15 IO: time series, radial profiles and spectra . . . 272
10.16 IO: graphic files, movie files, coeff files and potential files . . . 295
10.17 IO: RMS force balance, torsional oscillations, misc . . . 320
10.18 Reading and storing check points (restart files) . . . 358
10.19 Useful additional libraries . . . 364

Fortran Module Index **371**

Python Module Index **373**

Index **375**

CHAPTER

ONE

INTRODUCTION

1.1 Foreword

MagIC is a numerical code that can simulate fluid dynamics in a spherical shell. MagIC solves for the Navier-Stokes equation including Coriolis force, optionally coupled with an induction equation for Magneto-Hydro Dynamics (MHD) and a temperature (or entropy) equation under both the anelastic and the Boussinesq approximations.

MagIC uses Chebyshev polynomials in the radial direction and spherical harmonic decomposition in the azimuthal and latitudinal directions. The time-stepping scheme relies on a semi-implicit Crank-Nicolson for the linear terms of the MHD equations and a Adams-Bashforth scheme for the non-linear terms and the Coriolis force.

MagIC is written in Fortran and designed to be used on supercomputing clusters. It thus relies on a hybrid parallelisation scheme using both OpenMP and MPI. Postprocessing functions written in python (requiring matplotlib and scipy) are also provided to allow a useful data analysis.

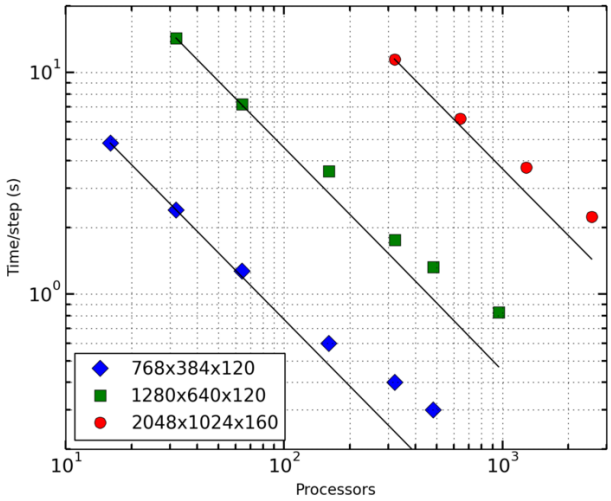

Fig. 1.1: Mean walltime of the MagIC code on the supercomputer stampede versus number of CPUs for a dynamo model computed at three different numerical resolutions (N_ϕ, N_θ, N_r). The solid black lines show the ideal scalings.

MagIC is a free software. It can be used, modified and redistributed under the terms of the GNU GPL v3 licence.

1.2 Giving credit

In case you intend to publish scientific results obtained with the MagIC code or present them in a conference, we (the developers of MagIC) kindly ask to be acknowledged with a reference to the website https://magic-sph.github.io/ or https://github.com/magic-sph/magic.

We also suggest to give appropriate reference to one or several of the following papers:

- Boussinesq equations: Wicht (2002, PEPI, 132, 281-302)
- Anelastic equations: Gastine & Wicht (2012, Icarus, 219, 28-442)
- Boussinesq benchmark: Christensen et al. (2001, PEPI, 128, 25-34)
- Anelastic benchmark: Jones et al. (2011, Icarus, 216, 120-135)
- In case you use the SHTns library for the spherical harmonics transforms (MagIC 5.3 or later), please also cite: Schaeffer (2013, GGG, 14, 751-758)

See also:

A (tentative) comprehensive list of the publications that have been produced to date (october 2015) using **MagIC** is accessible here. To date, more than **70 publications** have been-accepted in more than 10 different peer-reviewed journals: PEPI (19), Icarus (10), E&PSL (6), GJI (6), A&A (4), GRL (4), JFM (3), GAFD (3), Nature (2), etc.

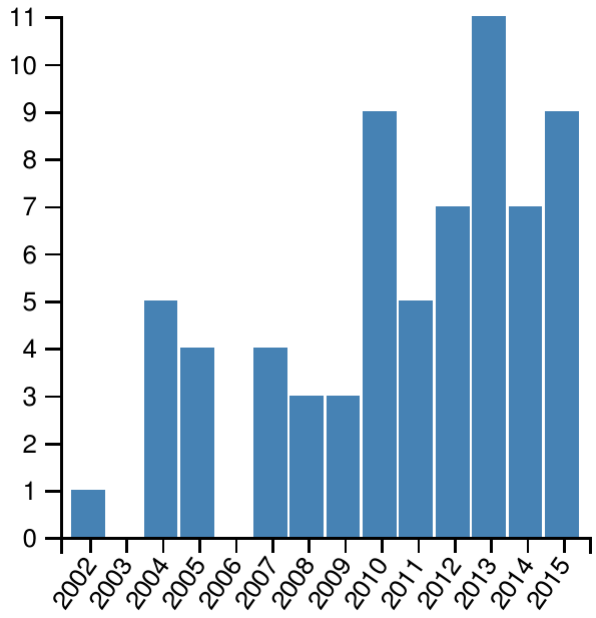

Fig. 1.2: Number of peer-reviewed publications produced using **MagIC**

CHAPTER

TWO

GET MAGIC AND RUN IT

2.1 Download the code

You can download a snapshot of the code from the Git repository using

```
$ git clone https://github.com/magic-sph/magic.git
```

In case you already have an account on github.com and uploaded a public SSH key on it, you could then rather use SSH:

```
$ git clone ssh://git@github.com/magic-sph/magic.git
```

2.2 Setting up the environment variables

Although not mandatory, it is strongly recommended to correctly source the environment variables of the MagIC code. It will ensure a smoother usage of the post-processing *python classes* and allow to run the *auto-test suite*. To do that, just go to the root directory of the MagIC code (`magic`) and source `sourceme` file that corresponds to your `$SHELL` environment variable.

In case you use bash, ksh or zsh, just use:

```
$ source sourceme.sh
```

In case you use csh or tcsh, rather use

```
$ source sourceme.csh
```

You can make sure that the environment variables have been correctly sourced by typing:

```
$ echo $MAGIC_HOME
$ echo $PYTHONPATH
```

If you don't want to `source sourceme.[c]sh` on each session, you can add the following into your `.bash_profile` (or `.profile` or `.zprofile` or `.cshrc`):

```
$ source whereverYouCheckedOut/magic/sourceme.sh
```

To get started, you then need to compile the code

2.3 Setting up compiler options and compiling

The **recommended way of compiling MagIC** is to use the build system CMake , if available on your platform. Otherwise, a backup solution is provided via the manual edition of a `Makefile`.

2.3.1 Generic compiling options

For both build systems (cmake or make), several build options can be toggled using the following available options:

- `PRECISION` Set it to 'dble' for double-precision calculations or to 'sngl' for single-precision calculations
- `OUT_PREC` Set it to 'dble' for double-precision in binary outputs or to 'sngl' for single precision
- `USE_MPI` Set to `yes` to use MPI, set it to `no` if you want a serial version of the code .
- `USE_OMP` Set it to `yes` to use the hybrid version of the code, or to `no` for a pure MPI (or serial) version.
- `USE_PRECOND` Set to `yes` to perform some pre-conditioning of the matrices.
- `USE_FFTLIB` This option lets you select the library you want to use for Fast Fourier Transforms. This can be set to 'JW' or 'MKL'. 'JW' refers to the inbuilt library by **J** ohannes **W** icht, while 'MKL' refers to the Intel Math Kernel Library. Use 'JW' if you don't have Intel MKL installed.
- `USE_LAPACKLIB` This option allows you to select the library you want to use for LU factorisation. This can be set to 'JW', 'MKL' or 'LAPACK'. 'JW' refers to the built-in library, while 'MKL' refers to the Intel Math Kernel Library and 'LAPACK' to the Lapack library
- `USE_HDF5` Set to `yes` if you want the restart file to be written in the HDF5 format
- `USE_SHTNS` Set to `yes` to use SHTns library for spherical harmonics transforms. The helper script `install-shtns.sh` is available in the `bin` directory to help installing SHTns.
- `PRODRUN` Set it to `yes` for production run, `no` for debugging.
- `DEBUG` Set to `all` to enable the full debug flags. *While running in debugging mode, set* `PRODRUN` *to* `no`.

Warning: MagIC cannot run with openMP alone, therefore a configuration of the form `USE_MPI=no`, `USE_OMP=yes` will be overwritten to force `USE_OMP=no`

2.3.2 Using `CMake` (recommended)

CMake is a powerful tool that can automatically detects and finds the best appropriate configuration for your platform. To use it, you just need to create a directory where you want to build the sources. For instance:

```
$ mkdir $MAGIC_HOME/build
$ cd $MAGIC_HOME/build
```

In a second step, you might want to specify your C and Fortran compilers (in case you skip this step, CMake will look for compilers for you but it might pick up another compiler as the one you might have wanted). For instance, in case you want to use the Intel compilers, you can export the following environment variables

```
$ export FC=mpiifort
$ export CC=mpiicc
```

for bash/ksh/zsh users and

```
$ setenv FC=mpiifort
$ setenv CC=mpiicc
```

for csh/tcsh users. At this stage you should be ready to build the code. If you simply use:

```
$ cmake ..
```

CMake will try to use the best options available on your machine (for instance it will try to locate and link the Intel Math Kernel Library). Otherwise you can pass the aforementioned available options to CMake using the generic form *-DOPTION=value*. For instance, in case you want to make use of the built-in libraries of MagIC and want to disable OpenMP, simply use

```
$ cmake .. -DUSE_OMP=no -DUSE_FFTLIB=JW -DUSE_LAPACKLIB=JW
```

Once you're happy with your configuration, just compile the code:

```
$ make -j
```

The executable `magic.exe` should have been produced in the local directory.

If you want to recompile the code from scratch do

```
$ make clean
```

to remove all the files generated by the compiler.

Once the executable is built, you are now ready to run your first production run!

2.3.3 Using `make` (backup solution)

In case CMake is not available on your platform, it is still possible to compile the code directly. Go to the directory where the source files of MagIC are contained

```
$ cd $MAGIC_HOME/src
```

Select compiler

Edit the file named `Makefile` using your favourite editor and set a suitable compiler for your platform using the variable: `COMPILER = value`. The possible options are `intel`, `gnu` or `portland` compilers.

List of default compilers

Compiler Option	Normal	With MPI
intel	ifort, icc	mpiifort, mpiicc
gnu	gfortran, gcc	mpif90, mpicc
portland	pgf95, pgcc	mpif90, mpicc

Warning: In case you want to use intel but `mpiifort` and `mpiicc` are not available, you may also need to adapt the variables `COMP_MPFC` and `COMP_MPCC`.

Select compiling options

You can also modify the different compiling options by editing the values of the various parameters defined in the first lines of the `Makefile`. For instance, in case you want to make use of the built-in libraries and want to disable OpenMP, just define

```
USE_OMP=no
USE_FFTLIB=JW
USE_LAPACKLIB=JW
```

MPI_INCPATH

This variable sets the path for your MPI header file `mpif.h`. This is in general useless if you already use the MPI wrappers such as `mpiifort` or `mpif90` to compile the code. It might be however required to define this path for some compiler configurations: `MPI_INCPATH` is usually `/usr/include` or `/usr/include/mpi` and should be found by the `Makefile` automatically thanks to the command `mpif90 --showme:incdirs`. In case this doesn't work, you may need to specify this variable manually in the `Makefile`. On supercomputing clusters, this variable is in general not used.

Other compilers

If your available compilers are different from the options provided in the `Makefile`, then just create a new profile for your desired compiler by changing the options `COMP_FC` and `COMP_CC` for serial fortran and C compilers and `COMP_MPFC` and `COMP_MPCC` for the possible MPI wrappers.

Once you've set up your compiling options compile the code using

```
$ make -j
```

The compiler should then produce an executable named `magic.exe`.

If you want to recompile the code from scratch do

```
$ make clean
```

to remove all the files generated by the compiler.

Once the executable is built, you are now ready to run your first production run!

2.4 Preparing a production run

After building the executable, use one of the namelists provided in the `$MAGIC_HOME/samples` directory (called `input.nml`), adapt it to your physical problem (see *here* for an exhaustive description of the possible options) and run **MagIC** as follows:

- Running a serial version of the code (`USE_MPI=no` and `USE_OMP=no`):

```
$ ./magic.exe input.nml
```

- Running the code without OpenMP (`USE_MPI=yes` and `USE_OMP=no`) with `<n_mpi>` MPI ranks:

```
$ mpiexec -n <n_mpi> ./magic.exe input.nml
```

- Running the hybrid code (`USE_MPI=yes` and `USE_OMP=yes`) with `<n_mpi>` MPI ranks and `<n_omp>` OpenMP threads:

```
$ export OMP_NUM_THREAD = <n_omp>
$ export KMP_AFFINITY=verbose,granularity=core,compact,1
$ mpiexec -n <n_mpi> ./magic.exe input.nml
```

Note that the *n_r_max-1* must be a multiple of `<n_mpi>`, where *n_r_max* is the number of radial grid points (see *here*).

CHAPTER

THREE

FORMULATION OF THE (MAGNETO)-HYDRODYNAMICS PROBLEM

The general equations describing thermal convection and dynamo action of a rotating compressible fluid are the starting point from which the Boussinesq or the anelastic approximations are developed. In MagIC, we consider a spherical shell rotating about the vertical z axis with a constant angular velocity Ω. Equations are solve in the corotating system.

The conservation of momentum is formulated by the Navier-Stokes equation

$$\rho\left(\frac{\partial \boldsymbol{u}}{\partial t}+\boldsymbol{u}\cdot\boldsymbol{\nabla}\,\boldsymbol{u}\right)=-\boldsymbol{\nabla}p+\frac{1}{\mu_0}(\boldsymbol{\nabla}\times\boldsymbol{B})\times\boldsymbol{B}+\rho\boldsymbol{g}-2\rho\boldsymbol{\Omega}\times\boldsymbol{u}+\boldsymbol{\nabla}\cdot\mathsf{S}, \tag{3.1}$$

where $\boldsymbol{u}$ is the velocity field, $\boldsymbol{B}$ the magnetic field, and p a modified pressure that includes centrifugal forces. S corresponds to the rate-of-strain tensor given by:

$$\begin{aligned} S_{ij}&=2\nu\rho\left[e_{ij}-\frac{1}{3}\delta_{ij}\,\boldsymbol{\nabla}\cdot\boldsymbol{u}\right],\\ e_{ij}&=\frac{1}{2}\left(\frac{\partial u_i}{\partial x_j}+\frac{\partial u_j}{\partial x_i}\right).\end{aligned}$$

Convection is driven by buoyancy forces acting on variations in density ρ. These variations have a dynamical part formulated by the continuity equation describing the conservation of mass:

$$\frac{\partial\rho}{\partial t}=-\boldsymbol{\nabla}\cdot\rho\boldsymbol{u}. \tag{3.2}$$

In addition an equation of state is required to formulate the thermodynamic density changes. For example the relation

$$\frac{1}{\rho}\partial\rho=-\alpha\partial T+\beta\partial p+\delta\partial\chi \tag{3.3}$$

describes density variations cause by variations in temperature T, pressure p, and composition χ. The latter contribution needs to be considered for describing the effects of light elements released from a growing solid iron core in a so-called double diffusive approach. Since this is not an option in the current versions of MagIC, we will neglect the respective contribution or model its effects with the co-density approach further discussed below.

To close the system we also have to formulate the dynamic changes of entropy, pressure, and composition. The evolution equation for pressure can be derived from the Navier-Stokes equation, as will be further discussed below. For entropy variations we use the so-called energy or heat equation

$$\rho T\left(\frac{\partial s}{\partial t}+\boldsymbol{u}\cdot\boldsymbol{\nabla}s\right)=\boldsymbol{\nabla}\cdot(k\boldsymbol{\nabla}T)+\Phi_\nu+\lambda\left(\boldsymbol{\nabla}\times\boldsymbol{B}\right)^2+\epsilon, \tag{3.4}$$

where Φ_ν corresponds to the viscous heating expressed by

$$\Phi_\nu=2\rho\left[e_{ij}e_{ji}-\frac{1}{3}\left(\boldsymbol{\nabla}\cdot\boldsymbol{u}\right)^2\right]$$

Note that we use here the summation convention over the indices i and j. The second last term on the right hand side is the Ohmic heating due to electric currents. The last term is a volumetric sink or source term that can describe various effects, for example radiogenic heating, the mixing-in of the light elements or, when radially dependent, potential variations in the adiabatic gradient (see below). Should compositional effects be considered an additional equivalent evolution equation for χ needs to be considered.

The induction equation is obtained from the Maxwell equations (ignoring displacement current) and Ohm's law (neglecting Hall effect):

$$\frac{\partial \boldsymbol{B}}{\partial t} = \boldsymbol{\nabla} \times (\boldsymbol{u} \times \boldsymbol{B} - \lambda \boldsymbol{\nabla} \times \boldsymbol{B}) \,. \tag{3.5}$$

When the magnetic diffusivity λ is homogeneous this simplifies to the commonly used form

$$\frac{\partial \boldsymbol{B}}{\partial t} = \boldsymbol{\nabla} \times (\boldsymbol{u} \times \boldsymbol{B}) + \lambda \boldsymbol{\Delta} \boldsymbol{B}. \tag{3.6}$$

The physical properties determining above equations are rotation rate Ω, the kinematic viscosity ν, the magnetic permeability μ_0, gravity $\boldsymbol{g}$, thermal conductivity k, magnetic diffusivity λ. The latter connects to the electrical conductivity σ via $\lambda = 1/(\mu_0 \sigma)$. The thermodynamics properties appearing in (3.3) are the thermal expansivity at constant pressure (and composition)

$$\alpha = \frac{1}{\rho} \left(\frac{\partial \rho}{\partial T} \right)_p \,, \tag{3.7}$$

the compressibility at constant temperature

$$\beta = \frac{1}{\rho} \left(\frac{\partial \rho}{\partial p} \right)_T$$

and an equivalent parameter δ for the dependence of density on composition.

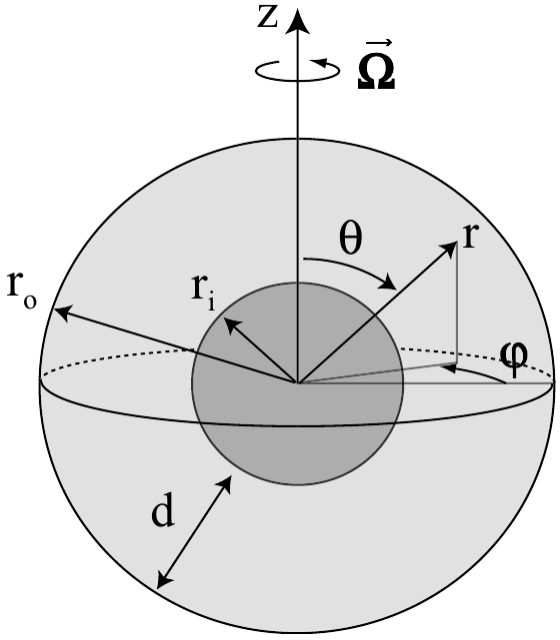

Fig. 3.1: Sketch of the spherical shell model and its system of coordinate.

3.1 The reference state

The convective flow and the related processes including magnetic field generation constitute only small disturbances around a background or reference state. In the following we denote the background state with a tilde and the disturbance we are interested in with a prime. Formally we will solve equations in first order of a smallness parameters ϵ which quantified the ratio of convective disturbances to background state:

$$\epsilon \sim \frac{T'}{\tilde{T}} \sim \frac{p'}{\tilde{p}} \sim \frac{\rho'}{\tilde{\rho}} \sim ... \ll 1. \tag{3.8}$$

The background state is hydrostatic, i.e. obeys the simple force balance

$$\nabla \tilde{p} = \tilde{\rho}\tilde{\boldsymbol{g}}. \tag{3.9}$$

Convective motions are supposed to be strong enough to provide homogeneous entropy (and composition). The reference state is thus adiabatic and its gradients can be expressed in terms of the pressure gradient (3.9):

$$\frac{\nabla \tilde{T}}{\tilde{T}} = \frac{1}{\tilde{T}} \left(\frac{\partial T}{\partial p} \right)_s \nabla p = \frac{\alpha}{c_p} \tilde{\boldsymbol{g}}, \tag{3.10}$$

$$\frac{\nabla \tilde{\rho}}{\tilde{\rho}} = \frac{1}{\tilde{\rho}} \left(\frac{\partial \rho}{\partial p} \right)_s \nabla p = \beta \tilde{\rho} \tilde{\boldsymbol{g}}. \tag{3.11}$$

The reference state obviously dependence only on radius. Dimensionless numbers quantifying the temperature and density gradients are called dissipation number Di and compressibility parameter Co respectively:

$$Di = \frac{\alpha d}{c_p} \tilde{g},$$

and

$$Co = d\beta\tilde{\rho}\tilde{g}.$$

Here d is a typical length scale, for example the shell thickness of the problem. The dissipation number is something like an inverse temperature scale hight while the compressibility parameters is an inverse density scale hight. The ratio of both numbers also helps to quantify the relative impact of temperature and pressure on density variations:

$$\frac{\alpha \nabla T}{\beta \nabla \rho} \approx \alpha \tilde{T} \frac{Di}{Co}. \tag{3.12}$$

As an example we demonstrate how to derive the first order continuity equation here. Using $\rho = \tilde{\rho} + \rho'$ in (3.2) leads to

$$\frac{\partial \tilde{\rho}}{\partial t} + \frac{\partial \rho'}{\partial t} = -\boldsymbol{\nabla} \cdot (\tilde{\rho}\boldsymbol{u}) - \boldsymbol{\nabla} \cdot (\rho'\boldsymbol{u}) \,.$$

The zero order term vanishes since the background density is considered static (or actually changing very slowly on very long time scales). The second term in the right hand side is obviously of second order. The ratio of the remaining two terms can be estimated to also be of first order in ϵ, meaning that the time derivative of ρ is actually also of second order:

$$\frac{[\partial \rho / \partial t]}{[\boldsymbol{\nabla} \cdot \rho \boldsymbol{u}]} \approx \frac{\rho'}{\tilde{\rho}} \approx \epsilon \quad .$$

Square brackets denote order of magnitude estimates here. We have used the fact that the reference state is static and assume time scale of changes are comparable (or slower) ρ' than the time scales represented by u and that length scales associated to the gradient operator are not too small. We can then neglect local variations in ρ' which means that sound waves are filtered out. This first order continuity equation thus simply reads:

$$\boldsymbol{\nabla} \cdot (\tilde{\rho}\boldsymbol{u}) = 0. \tag{3.13}$$

This defines the so-called anelastic approximation where sound waves are filtered out by neglecting the local time derivative of density. This approximation is justified when typical velocities are sufficiently smaller than the speed of sound.

3.2 Boussinesq approximation

For Earth the dissipation number and the compressibility parameter are around 0.2 when temperature and density jump over the whole liquid core are considered. This motivates the so called Boussinesq approximation where Di and Co are assumed to vanish. The continuity equation (3.2) then simplifies further:

$$\frac{1}{\tilde{\rho}}\boldsymbol{\nabla}\cdot\tilde{\rho}\boldsymbol{u}=\frac{\boldsymbol{u}}{\tilde{\rho}}\cdot\nabla\tilde{\rho}+\nabla\cdot\boldsymbol{u}\approx\nabla\cdot\boldsymbol{u}=0.$$

When using typical number for Earth, (3.12) becomes 0.05 so that pressure effects on density may be neglected. The first order Navier-Stokes equation (after to zero order hydrostatic reference solution has been subtracted) then reads:

$$\tilde{\rho}\left(\frac{\partial\boldsymbol{u}}{\partial t}+\boldsymbol{u}\cdot\boldsymbol{\nabla}\,\boldsymbol{u}\right)=-\boldsymbol{\nabla}p'-2\rho\boldsymbol{\Omega}\times\boldsymbol{u}+\alpha\tilde{g}_oT'\frac{\boldsymbol{r}}{r_o}+\frac{1}{\mu_0}(\boldsymbol{\nabla}\times\boldsymbol{B})\times\boldsymbol{B}+\tilde{\rho}\nu\Delta\boldsymbol{u}. \tag{3.14}$$

Here u and B are understood as first order disturbances and p' is the first order non-hydrostatic pressure and T' the super-adiabatic temperature. Above we have adopted a simplification of the buoyancy term. In the Boussinesq limit with vanishing Co and a small density difference between a solid inner and a liquid outer core a linear gravity dependence provides a reasonable approximation:

$$\tilde{\boldsymbol{g}}=\tilde{g}_o\frac{\boldsymbol{r}}{r_o},$$

where we have chosen the gravity $\tilde{g}_o$ at the outer boundary radius r_o as reference.

The first order energy equation becomes

$$\tilde{\rho}\left(\frac{\partial T'}{\partial t}+\boldsymbol{u}\cdot\boldsymbol{\nabla}T'\right)=\kappa\Delta T'+\epsilon, \tag{3.15}$$

where we have assumed a homogeneous k and neglected viscous and Ohmic heating which can be shown to scale with Di as we discuss below. Further more, we have used the simple relation

$$\partial s\approx\frac{\tilde{\rho}c_p}{\tilde{T}}\partial T,$$

defined the thermal diffusivity

$$\kappa=\frac{k}{\tilde{\rho}c_p},$$

and adjusted the definition of ϵ.

MagIC solves a dimensionless form of the differential equations. Time is scaled in units of the viscous diffusion time d^2/ν, length in units of the shell thickness d, temperature in units of the temperature drop $\Delta T=T_o-T_i$ over the shell, and magnetic field in units $(\mu\lambda\tilde{\rho}\Omega)^{1/2}$. Technically the transition to the dimensionless form is achieved by the substitution

$$r\to r\;d,t\to(d^2/\nu)\;t,T\to\Delta T\;T,B\to(\mu\lambda\tilde{\rho}\Omega)^{1/2}\;B$$

where r stands for any length. The next step then is to collect the physical properties as few a possible characteristic dimensionless numbers. Note that many different scalings and combinations of dimensionless numbers are possible. For the Navier-Stokes equation in the Boussinesq limit MagIC uses the form:

$$\left(\frac{\partial\boldsymbol{u}}{\partial t}+\boldsymbol{u}\cdot\boldsymbol{\nabla}\,\boldsymbol{u}\right)=-\boldsymbol{\nabla}p'-\frac{2}{E}\boldsymbol{e_z}\times\boldsymbol{u}+\frac{Ra}{Pr}T'\frac{\boldsymbol{r}}{r_o}+\frac{1}{EPm}(\boldsymbol{\nabla}\times\boldsymbol{B})\times\boldsymbol{B}+\Delta\boldsymbol{u}, \tag{3.16}$$

where $\boldsymbol{e}_z$ is the unit vector in the direction of the rotation axis and the meaning of the pressure disturbance p' has been adjusted to the new dimensionless equation form.

3.3 Anelastic approximation

The anelastic approximation adopts the simplified continuity (3.13). The background state can be specified in different ways, for example by providing profiles based on internal models and/or ab initio simulations. We will assume a polytropic ideal gas in the following.

3.3.1 Analytical solution in the limit of an ideal gas

In the limit of an ideal gas which follows $\tilde{p} = \tilde{\rho}\tilde{T}$ and has $\alpha = 1/\tilde{T}$, one directly gets:

$$\begin{aligned} \frac{d\tilde{T}}{dr} &= -Di\,\tilde{g}(r), \\ \tilde{\rho} &= \tilde{T}^{1/(\gamma-1)}, \end{aligned}$$

where $\gamma = c_p/c_v$. Note that we have moved to a dimensionless formulations here, where all quantities have been normalized with their outer boundary values and Di refers to the respective outer boundary value. If we in addition make the assumption of a centrally-condensed mass in the center of the spherical shell of radius $r \in [r_i, r_o]$, i.e. $g \propto, 1/r^2$, this leads to

$$\begin{aligned} \tilde{T}(r) &= Di\frac{r_o^2}{r} + (1 - Di\,r_o), \\ \tilde{\rho}(r) &= \tilde{T}^m, \\ Di &= \frac{r_i}{r_o}\left(\exp\frac{N_\rho}{m} - 1\right), \end{aligned}$$

where $N_\rho = \ln(\tilde{\rho}_i/\tilde{\rho}_o)$ is the number of density scale heights of the reference state and $m = 1/(\gamma-1)$ is the polytropic index.

Warning:

- The relationship between N_ρ and the dissipation number Di directly depends on the gravity profile. The formula above is only valid when $g \propto 1/r^2$.
- In this formulation, when you change the polytropic index m, you also change the nature of the fluid you're modelling since you accordingly modify $\gamma = c_p/c_v$.

3.3.2 Anelastic MHD equations

In the most general formulation, all physical properties defining the background state may vary with depth. Specific reference values must then be chosen to provide a unique dimensionless formulations and we typically chose outer boundary values here. The exception is the magnetic diffusivity where we adopt the inner boundary value instead. The motivation is twofold: (i) it allows an easier control of the possible continuous conductivity value in the inner core; (ii) it is a more natural choice when modelling gas giants planets which exhibit a strong electrical conductivity decay in the outer layer.

The time scale is then the viscous diffusion time d^2/ν_o where ν_o is the kinematic viscosity at the outer boundary. Magnetic field is expressed in units of $(\rho_o\mu_0\lambda_i\Omega)^{1/2}$, where ρ_o is the density at the outer boundary and λ_i is the magnetic diffusivity at the **inner** boundary.

This leads to the following sets of dimensionless equations:

$$\left(\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u}\cdot\boldsymbol{\nabla}\boldsymbol{u}\right) = -\boldsymbol{\nabla}\left(\frac{p'}{\tilde{\rho}}\right) - \frac{2}{E}\boldsymbol{e_z}\times\boldsymbol{u} + \frac{Ra}{Pr}\tilde{g}\,s'\,\boldsymbol{e_r} + \frac{1}{Pm\,E\,\tilde{\rho}}(\boldsymbol{\nabla}\times\boldsymbol{B})\times\boldsymbol{B} + \frac{1}{\tilde{\rho}}\boldsymbol{\nabla}\cdot\mathsf{S}, \tag{3.17}$$

$$\boldsymbol{\nabla} \cdot \tilde{\rho} \boldsymbol{u} = 0, \tag{3.18}$$

$$\boldsymbol{\nabla} \cdot \boldsymbol{B} = 0, \tag{3.19}$$

$$\frac{\partial \boldsymbol{B}}{\partial t} = \boldsymbol{\nabla} \times (\boldsymbol{u} \times \boldsymbol{B}) - \frac{1}{Pm} \boldsymbol{\nabla} \times (\lambda(r) \boldsymbol{\nabla} \times \boldsymbol{B}) . \tag{3.20}$$

Here $\tilde{g}$ and $\tilde{\rho}$ are the normalized radial gravity and density profiles that reach one at the outer boundary.

3.3.3 Entropy equation and turbulent diffusion

The entropy equation usually requires an additional assumption in most of the existing anelastic approximations. Indeed, if one simply expands Eq. (3.4) with the classical temperature diffusion an operator of the form:

$$\epsilon \boldsymbol{\nabla} \cdot (K \boldsymbol{\nabla} T') + \boldsymbol{\nabla} \cdot \left(K \boldsymbol{\nabla} \tilde{T} \right),$$

will remain the right-hand side of the equation. At first glance, there seems to be a $1/\epsilon$ factor between the first term and the second one, which would suggest to keep only the second term in this expansion. However, for astrophysical objects which exhibit strong convective driving (and hence large Rayleigh numbers), the diffusion of the adiabatic background is actually very small and may be comparable or even smaller in magnitude than the ϵ terms representing the usual convective perturbations. For the Earth core for instance, if one assumes that the typical temperature fluctuations are of the order of 1 mK and the temperature contrast between the inner and outer core is of the order of 1000 K, then $\epsilon \sim 10^{-6}$. The ratio of the two terms can thus be estimated as

$$\epsilon \frac{T'/\delta^2}{T/d^2}, \tag{3.21}$$

where d is the thickness of the inner core and δ is the typical thermal boundary layer thickness. This ratio is exactly one when $\delta = 1$ m, a plausible value for the Earth inner core.

In numerical simulations however, the over-estimated diffusivities restrict the computational capabilities to much lower Rayleigh numbers. As a consequence, the actual boundary layers in a global DNS will be much thicker and the ratio (3.21) will be much smaller than unity. The second terms will thus effectively acts as a radial-dependent heat source or sink that will drive or hinder convection. This is one of the physical motivation to rather introduce a **turbulent diffusivity** that will be approximated by

$$\kappa \tilde{\rho} \tilde{T} \boldsymbol{\nabla} s,$$

where κ is the turbulent diffusivity. **Entropy diffusion is assumed to dominate over temperature diffusion in turbulent flows**.

The choice of the entropy scale to non-dimensionalize Eq. (3.4) also depends on the nature of the boundary conditions: it can be simply the entropy contrast over the layer Δs when the entropy is held constant at both boundaries, or $d\,(ds/dr)$ when flux-based boundary conditions are employed. We will restrict to the first option in the following, but keep in mind that depending on your setup, the entropy reference scale (and thus the Rayleigh number definition) might change.

$$\tilde{\rho} \tilde{T} \left(\frac{\partial s'}{\partial t} + \boldsymbol{u} \cdot \boldsymbol{\nabla} s' \right) = \frac{1}{Pr} \boldsymbol{\nabla} \cdot \left(\kappa(r) \tilde{\rho} \tilde{T} \boldsymbol{\nabla} s' \right) + \frac{Pr\, Di}{Ra} \Phi_\nu + \frac{Pr\, Di}{Pm^2\, E\, Ra} \lambda(r) \left(\boldsymbol{\nabla} \times \boldsymbol{B} \right)^2 , \tag{3.22}$$

A comparison with (3.17) reveals meaning of the different non-dimensional numbers that scale viscous and Ohmic heating. The fraction Pr/Ra simply expresses the ratio of entropy and flow in the Navier-Stokes equation, while the additional factor $1/EPm$ reflects the scale difference of magnetic field and flow. Then remaining dissipation number Di then expresses the relative importance of viscous and Ohmic heating compared to buoyancy and Lorentz force in the Navier-Stokes equation. For small Di both heating terms can be neglected compared to entropy changes due to advection, an limit that is used in the Boussinesq approximation.

3.4 Dimensionless control parameters

The equations (3.17)-(3.22) are governed by four dimensionless numbers: the Ekman number

$$E = \frac{\nu}{\Omega d^2}, \tag{3.23}$$

the Rayleigh number

$$Ra = \frac{\alpha_o g_o T_o d^3 \Delta s}{c_p \kappa_o \nu_o}, \tag{3.24}$$

the Prandtl number

$$Pr = \frac{\nu_o}{\kappa_o}, \tag{3.25}$$

and the magnetic Prandtl number

$$Pm = \frac{\nu_o}{\lambda_i}. \tag{3.26}$$

In addition to these four numbers, the reference state is controlled by the geometry of the spherical shell given by its radius ratio

$$\eta = \frac{r_i}{r_o}, \tag{3.27}$$

and the background density and temperature profiles, either controlled by Di or by N_ρ and m.

In the Boussinesq approximation all physical properties are assumed to be homogeneous and we can drop the sub-indices o and i except for gravity. Moreover, the Rayleigh number can be expressed in terms of the temperature jump across the shell:

$$Ra = \frac{\alpha g_o d^3 \Delta T}{\kappa \nu}. \tag{3.28}$$

See also:

In MagIC, those control parameters can be adjusted in the *&phys_param* section of the input namelist.

Variants of the non-dimensional equations and control parameters result from different choices for the fundamental scales. For the length scale often r_o is chosen instead of d. Other natural scales for time are the magnetic or the thermal diffusion time, or the rotation period. There are also different options for scaling the magnetic field strength. The prefactor of two, which is retained in the Coriolis term in (3.17), is often incorporated into the definition of the Ekman number.

See also:

Those references timescales and length scales can be adjusted by several input parameters in the *&control* section of the input namelist.

3.4.1 Usual diagnostic quantities

Characteristic properties of the solution are usually expressed in terms of non-dimensional diagnostic parameters. In the context of the geodynamo for instance, the two most important ones are the magnetic Reynolds number Rm and the Elsasser number Λ. Usually the rms-values of the velocity u_{rms} and of the magnetic field B_{rms} inside the spherical shell are taken as characteristic values. The magnetic Reynolds number

$$Rm = \frac{u_{rms} d}{\lambda_i}$$

can be considered as a measure for the flow velocity and describes the ratio of advection of the magnetic field to magnetic diffusion. Other characteristic non-dimensional numbers related to the flow velocity are the (hydrodynamic) Reynolds number

$$Re = \frac{u_{rms} d}{\nu_o},$$

which measures the ratio of inertial forces to viscous forces, and the Rossby number

$$Ro = \frac{u_{rms}}{\Omega d},$$

a measure for the ratio of inertial to Coriolis forces.

$$\Lambda = \frac{B_{rms}^2}{\mu_0 \lambda_i \rho_o \Omega}$$

measures the ratio of Lorentz to Coriolis forces and is equivalent to the square of the non-dimensional magnetic field strength in the scaling chosen here.

See also:

The time-evolution of these diagnostic quantities are stored in the *par.TAG* file produced during the run of MagIC.

3.5 Boundary conditions and treatment of inner core

3.5.1 Mechanical conditions

In its simplest form, when modelling the geodynamo, the fluid shell is treated as a container with rigid, impenetrable, and co-rotating walls. This implies that within the rotating frame of reference all velocity components vanish at r_o and r_i. In case of modelling the free surface of a gas giant planets or a star, it is preferable to rather replace the condition of zero horizontal velocity by one of vanishing viscous shear stresses (the so-called free-slip condition).

Furthermore, even in case of modelling the liquid iron core of a terrestrial planet, there is no a priori reason why the inner core should necessarily co-rotate with the mantle. Some models for instance allow for differential rotation of the inner core and mantle with respect to the reference frame. The change of rotation rate is determined from the net torque. Viscous, electromagnetic, and torques due to gravitational coupling between density heterogeneities in the mantle and in the inner core contribute.

See also:

The mechanical boundary conditions can be adjusted with the parameters *ktopv* and *kbotv* in the *&phys_param* section of the input namelist.

3.5.2 Magnetic boundary conditions and inner core conductivity

When assuming that the fluid shell is surrounded by electrically insulating regions (inner core and external part), the magnetic field inside the fluid shell matches continuously to a potential field in both the exterior and the interior regions. Alternative magnetic boundary conditions (like cancellation of the horizontal component of the field) are also possible.

Depending on the physical problem you want to model, treating the inner core as an insulator is not realistic either, and it might instead be more appropriate to assume that it has the same electrical conductivity as the fluid shell. In this case, an equation equivalent to (3.20) must be solved for the inner core, where the velocity field simply describes the solid body rotation of the inner core with respect to the reference frame. At the inner core boundary a continuity condition for the magnetic field and the horizontal component of the electrical field apply.

See also:

The magnetic boundary conditions can be adjusted with the parameters *ktopb* and *kbotb* in the *&phys_param* section of the input namelist.

3.5.3 Thermal boundary conditions and distribution of buoyancy sources

In many dynamo models, convection is simply driven by an imposed fixed super-adiabatic entropy contrast between the inner and outer boundaries. This approximation is however not necessarily the best choice, since for instance, in the present Earth, convection is thought to be driven by a combination of thermal and compositional buoyancy. Sources of heat are the release of latent heat of inner core solidification and the secular cooling of the outer and inner core, which can effectively be treated like a heat source. The heat loss from the core is controlled by the convecting mantle, which effectively imposes a condition of fixed heat flux at the core-mantle boundary on the dynamo. The heat flux is in that case spatially and temporally variable.

See also:

The thermal boundary conditions can be adjusted with the parameters *ktops* and *kbots* in the *&phys_param* section of the input namelist.

CHAPTER

FOUR

NUMERICAL TECHNIQUE

MagIC is a pseudo-spectral MHD code. This numerical technique was originally developed by P. Gilman and G. Glatzmaier for the spherical geometry. In this approach the unknowns are expanded into complete sets of functions in radial and angular directions: **Chebyshev polynomials in the radial direction** and **spherical harmonic functions in the azimuthal and latitudinal directions**. This allows to express all partial derivatives analytically. Employing orthogonality relations of spherical harmonic functions and using collocation in radius then lead to algebraic equations that are integrated in time with a **mixed implicit/explicit time stepping scheme**. The nonlinear terms and the Coriolis force are evaluated in the physical (or grid) space rather than in spectral space. Although this approach requires costly numerical transformations between the two representations (from spatial to spectral using Legendre and Fourier transforms), the resulting decoupling of all spherical harmonic modes leads to a net gain in computational speed. Before explaining these methods in more detail, we introduce the poloidal/toroidal decomposition.

4.1 Poloidal/toroidal decomposition

Any vector $\boldsymbol{v}$ that fulfills $\boldsymbol{\nabla}\cdot\boldsymbol{v}=0$, i.e. a so-called *solenoidal field*, can be decomposed in a poloidal and a toroidal part W and Z, respectively

$$\boldsymbol{v}=\boldsymbol{\nabla}\times(\boldsymbol{\nabla}\times W\,\boldsymbol{e_r})+\boldsymbol{\nabla}\times Z\,\boldsymbol{e_r}.$$

Three unknown vector components are thus replaced by two scalar fields, the poloidal potential W and the toroidal potential Z. This decomposition is unique, aside from an arbitrary radial function $f(r)$ that can be added to W or Z without affecting $\boldsymbol{v}$.

In the anelastic approximation, such a decomposition can be used for the mass flux $\tilde{\rho}\boldsymbol{u}$ and the magnetic field $\boldsymbol{B}$. This yields

$$\boxed{\begin{aligned}\tilde{\rho}\boldsymbol{u}&=\boldsymbol{\nabla}\times(\boldsymbol{\nabla}\times W\,\boldsymbol{e_r})+\boldsymbol{\nabla}\times Z\,\boldsymbol{e_r}\,,\\ \boldsymbol{B}&=\boldsymbol{\nabla}\times(\boldsymbol{\nabla}\times g\,\boldsymbol{e_r})+\boldsymbol{\nabla}\times h\,\boldsymbol{e_r}\,.\end{aligned}} \tag{4.1}$$

The two scalar potentials of a divergence free vector field can be extracted from its radial component and the radial component of its curl using the fact that the toroidal field has not radial component:

$$\begin{aligned}\boldsymbol{e_r}\cdot\tilde{\rho}\boldsymbol{u}&=-\Delta_H\,W,\\ \boldsymbol{e_r}\cdot(\boldsymbol{\nabla}\times\boldsymbol{u})&=-\Delta_H\,Z,\end{aligned} \tag{4.2}$$

where the operator Δ_H denotes the horizontal part of the Laplacian:

$$\Delta_H=\frac{1}{r^2\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial}{\partial\theta}\right)+\frac{1}{r^2\sin^2\theta}\frac{\partial^2}{\partial^2\phi}. \tag{4.3}$$

The equation (4.1) can be expanded in spherical coordinates. The three components of $\tilde{\rho}\boldsymbol{u}$ are given by

$$\tilde{\rho}\boldsymbol{u}=-(\Delta_H W)\,\boldsymbol{e_r}+\left(\frac{1}{r}\frac{\partial W}{\partial\theta}+\frac{1}{r\sin\theta}\frac{\partial Z}{\partial\phi}\right)\boldsymbol{e_\theta}+\left(\frac{1}{r\sin\theta}\frac{\partial W}{\partial\phi}-\frac{1}{r}\frac{\partial Z}{\partial\theta}\right)\boldsymbol{e_\phi}, \tag{4.4}$$

while the curl of $\tilde{\rho}\boldsymbol{u}$ is expressed by

$$\begin{aligned}\boldsymbol{\nabla}\times\tilde{\rho}\boldsymbol{u} = &-\left(\Delta_H Z\right)\boldsymbol{e_r} + \left[-\frac{1}{r\sin\theta}\frac{\partial}{\partial\phi}\left(\frac{\partial^2}{\partial r^2}+\Delta_H\right)W + \frac{1}{r}\frac{\partial^2 Z}{\partial r\partial\theta}\right]\boldsymbol{e_\theta}\\ &+\left[\frac{1}{r}\frac{\partial}{\partial\theta}\left(\frac{\partial^2}{\partial r^2}+\Delta_H\right)W + \frac{1}{r\sin\theta}\frac{\partial^2 Z}{\partial r\partial\phi}\right]\boldsymbol{e_\phi},\end{aligned} \tag{4.5}$$

Using the horizontal part of the divergence operator

$$\boldsymbol{\nabla}_H = r\sin\frac{\partial(\sin\theta}{\partial\theta}\,\boldsymbol{e}_\theta + r\sin\frac{\partial}{\partial\phi}\,\boldsymbol{e}_\phi$$

above expressions can be simplified to

$$\tilde{\rho}\boldsymbol{u} = -\Delta_H\,\boldsymbol{e_r}\,W + \boldsymbol{\nabla}_H\frac{\partial}{\partial r}\,W + \boldsymbol{\nabla}_H\times\boldsymbol{e}_r\,Z$$

and

$$\nabla\times\tilde{\rho}\boldsymbol{u} = -\Delta_H\,\boldsymbol{e}_r\,Z + \boldsymbol{\nabla}_H\frac{\partial}{\partial r}\,Z - \boldsymbol{\nabla}_H\times\Delta_H\boldsymbol{e}_r\,W\quad .$$

Below we will use the fact that the horizontal components of the poloidal field depend on the radial derivative of the poloidal potential.

4.2 Spherical harmonic representation

Spherical harmonic functions Y_ℓ^m are a natural choice for the horizontal expansion in colatitude θ and longitude ϕ:

$$Y_\ell^m(\theta,\phi) = P_\ell^m(\cos\theta)\,e^{im\phi},$$

where ℓ and m denote spherical harmonic degree and order, respectively, P_ℓ^m is an associated Legendre function. Different normalization are in use. Here we adopt a complete normalization so that the orthogonality relation reads

$$\int_0^{2\pi} d\,\phi\int_0^{\pi}\sin\theta\,d\theta\;Y_\ell^m(\theta,\phi)\,Y_{\ell'}^{m'}(\theta,\phi)\;=\;\delta_{\ell\ell'}\delta^{mm'}. \tag{4.6}$$

This means that

$$Y_\ell^m(\theta,\phi) = \sqrt{\frac{1}{2\pi}\frac{(2\ell+1)(\ell-|m|)!}{2(\ell+|m|)!}}P_\ell^m(\cos\theta)\,e^{im\phi}\,(-1)^m,$$

As an example, the spherical harmonic representation of the magnetic poloidal potential $g(r,\theta,\phi)$, truncated at degree and order ℓ_{max}, then reads

$$g(r,\theta,\phi) = \sum_{\ell=0}^{\ell_{max}}\sum_{m=-\ell}^{\ell} g_{\ell m}(r)\,Y_\ell^m(\theta,\phi), \tag{4.7}$$

with

$$g_{\ell m}(r) = \frac{1}{\pi}\int_0^{\pi} d\theta\sin\theta\;g_m(r,\theta)\;P_\ell^m(\cos\theta), \tag{4.8}$$

$$g_m(r,\theta) = \frac{1}{2\pi}\int_0^{2\pi} d\phi\;g(r,\theta,\phi)\;e^{-im\phi}. \tag{4.9}$$

The potential $g(r,\theta,\phi)$ is a real function so that $g^\star_{\ell m}(r) = g_{\ell,-m}(r)$, where the asterisk denotes the complex conjugate. Thus, only coefficients with $m \geq 0$ have to be considered. The same kind of expansion is made for the toroidal magnetic potential, the mass flux potentials, pressure and entropy (or temperature).

The equations (4.8) and (4.9) define a two-step transform from the longitude/latitude representation to the spherical harmonic representation $(r,\theta,\phi) \longrightarrow (r,\ell,m)$. The equation (4.7) formulates the inverse procedure $(r,\ell,m) \longrightarrow (r,\theta,\phi)$. Fast-Fourier transforms are employed in the longitudinal direction, requiring (at least) $N_\phi = 2\ell_{max}+1$ evenly spaced grid points ϕ_i. MagIC relies on the Gauss-Legendre quadrature for evaluating the integral (4.8)

$$g_{\ell m}(r) = \frac{1}{N_\theta}\sum_{j=1}^{N_\theta} w_j\, g_m(r,\theta_j)\; P_\ell^m(\cos\theta_j),$$

where θ_j are the N_θ Gaussian quadrature points defining the latitudinal grid, and w_j are the respective weights. Pre-stored values of the associated Legendre functions at grid points θ_j in combination with a FFT in ϕ provide the inverse transform (4.7). Generally, $N_\phi = 2N_\theta$ is chosen, which provides isotropic resolution in the equatorial region. Choosing $\ell_{max} = [\min(2N_\theta, N_\phi) - 1]/3$ prevents aliasing errors.

See also:

In MagIC, the Legendre functions are defined in the subroutine `plm_theta`. The Legendre transforms from spectral to grid space are computed in the module `legendre_spec_to_grid`, while the backward transform (from grid space to spectral space) are computed in the module `legendre_grid_to_spec`. The fast Fourier transforms are computed in the module `fft`.

4.2.1 Special recurrence relations

The action of a horizontal Laplacian (4.3) on spherical harmonics can be analytically expressed by

$$\boxed{\Delta_H Y_\ell^m = -\frac{\ell(\ell+1)}{r^2}\, Y_\ell^m\,.} \tag{4.10}$$

They are several useful recurrence relations for the Legendre polynomials that will be further employed to compute Coriolis forces and the θ and ϕ derivatives of advection and Lorentz forces. Four of these operators are used in **MagIC**. The first one is defined by

$$\vartheta_1 = \frac{1}{\sin\theta}\frac{\partial}{\partial\theta}\sin^2\theta = \sin\theta\frac{\partial}{\partial\theta} + 2\cos\theta\,.$$

The action of this operator on a Legendre polynomials is given by

$$\vartheta_1 = (\ell+2)\, c_{\ell+1}^m\, P_{\ell+1}^m(\cos\theta) - (\ell-1)\, c_\ell^m\, P_{\ell-1}^m(\cos\theta),$$

where c_ℓ^m is defined by

$$c_\ell^m = \sqrt{\frac{(\ell+m)(\ell-m)}{(2\ell-1)(2\ell+1)}}\,. \tag{4.11}$$

How is that implemented in the code? Let's assume we want the spherical harmonic contribution of degree ℓ and order *m* for the expression

$$\frac{1}{\sin\theta}\frac{\partial}{\partial\theta}(\sin\theta\, f(\theta))\,.$$

In order to employ the operator ϑ_1 for the derivative, we thus define a new function

$$F(\theta) = f(\theta)/\sin\theta\,,$$

so that

$$\frac{1}{\sin\theta}\frac{\partial}{\partial\theta}[\sin\theta\, f(\theta)] = \vartheta_1 F(\theta)\,.$$

Expanding $F(\theta)$ in Legendre polynomials and using the respective orthogonality relation we can then map out the required contribution in the following way:

$$\boxed{\int_0^\pi d\theta\, \sin\theta\, P_\ell^m \vartheta_1 \sum_{\ell'} F_{\ell'}^m P_{\ell'}^m = (\ell+1)\, c_\ell^m\, F_{\ell-1}^m - \ell\, c_{\ell+1}^m\, F_{\ell+1}^m\,.} \tag{4.12}$$

Here, we have assumed that the Legendre functions are completely normalized such that

$$\int_0^\pi d\theta\, \sin\theta\, P_\ell^m P_{\ell'}^m = \delta_{\ell\ell'}\,.$$

See also:

This operator is defined in the module `horizontal_data` by the variables `dTheta1S` for the first part of the right-hand side of (4.12) and `dTheta1A` for the second part.

The second operator used to formulate colatitude derivatives is

$$\vartheta_2 = \sin\theta\frac{\partial}{\partial\theta}\,.$$

The action of this operator on the Legendre polynomials reads

$$\vartheta_2 P_\ell^m(\cos\theta) = \ell\, c_{\ell+1}^m\, P_{\ell+1}^m(\cos\theta) - (\ell+1)\, c_\ell^m\, P_{\ell-1}^m(\cos\theta)\,,$$

so that

$$\boxed{\int_0^\pi d\theta\, \sin\theta\, P_\ell^m \vartheta_2 \sum_{\ell'} f_{\ell'}^m P_{\ell'}^m = (\ell-1)\, c_\ell^m\, f_{\ell-1}^m - (\ell+2)\, c_{\ell+1}^m\, f_{\ell+1}^m\,.} \tag{4.13}$$

See also:

This operator is defined in the module `horizontal_data` by the variables `dTheta2S` for the first part of the right-hand side of (4.13) and `dTheta2A` for the second part.

The third combined operator is defined by:

$$\vartheta_3 = \sin\theta\frac{\partial}{\partial\theta} + \cos\theta\, L_H\,,$$

where $-L_H/r^2 = \Delta_H$.

Acting with ϑ_3 on a Legendre function gives:

$$\vartheta_3 P_\ell^m(\cos\theta) = \ell(\ell+1)\, c_{\ell+1}^m\, P_{\ell+1}^m(\cos\theta) + (\ell-1)(\ell+1)\, c_\ell^m\, P_{\ell-1}^m(\cos\theta)\,,$$

which results into:

$$\boxed{\int_0^\pi d\theta\, \sin\theta\, P_\ell^m \vartheta_3 \sum_{\ell'} f_{\ell'}^m P_{\ell'}^m = (\ell-1)(\ell+1)\, c_\ell^m\, f_{\ell-1}^m + \ell(\ell+2)\, c_{\ell+1}^m\, f_{\ell+1}^m\,.} \tag{4.14}$$

See also:

This operator is defined in the module `horizontal_data` by the variables `dTheta3S` for the first part of the right-hand side of (4.14) and `dTheta3A` for the second part.

The fourth (and last) combined operator is defined by:

$$\vartheta_4 = \frac{1}{\sin\theta}\frac{\partial}{\partial\theta}\sin^2\theta\, L_H = \vartheta 1\, L_H\,,$$

Acting with ϑ_3 on a Legendre function gives:

$$\vartheta_4 P_\ell^m(\cos\theta) = \ell(\ell+1)(\ell+2)\, c_{\ell+1}^m\, P_{\ell+1}^m(\cos\theta) - \ell(\ell-1)(\ell+1)\, c_\ell^m\, P_{\ell-1}^m(\cos\theta)\,,$$

which results into:

$$\boxed{\int_0^\pi d\theta\, \sin\theta\, P_\ell^m \vartheta_4 \sum_{\ell'} f_{\ell'}^m P_{\ell'}^m = \ell(\ell-1)(\ell+1)\, c_\ell^m\, f_{\ell-1}^m - \ell(\ell+1)(\ell+2)\, c_{\ell+1}^m\, f_{\ell+1}^m\,.} \tag{4.15}$$

See also:

This operator is defined in the module `horizontal_data` by the variables `dTheta4S` for the first part of the right-hand side of (4.15) and `dTheta4A` for the second part.

4.3 Radial representation

In MagIC, the radial dependencies are expanded into complete sets of functions: the Chebyshev polynomials $\mathcal{C}(x)$. The polynomial of degree n is defined by

$$\mathcal{C}_n(x) = \cos\left[n\, \arccos(x)\right] \quad -1 \le x \le 1\,.$$

When truncating at degree N, the radial expansion of the poloidal magnetic potential reads

$$g_{\ell m}(r) = \sum_{n=0}^{N} g_{\ell m n}\, \mathcal{C}_n(r), \tag{4.16}$$

with

$$g_{\ell m n} = \frac{2-\delta_{n0}}{\pi}\int_{-1}^{1} \frac{dx\, g_{\ell m}(r(x))\, \mathcal{C}_n(x)}{\sqrt{1-x^2}}. \tag{4.17}$$

The Chebyshev definition space $(-1 \le x \le 1)$ is then linearly mapped onto a radius range $(r_i \le r \le r_o)$ by

$$x(r) = 2\frac{r-r_i}{r_o-r_i} - 1. \tag{4.18}$$

In addition, nonlinear mapping can be defined to modify the radial dependence of the grid-point density.

When choosing the N_r extrema of $\mathcal{C}_{N_r-1}$ as radial grid points,

$$x_k = \cos\left(\pi\frac{(k-1)}{N_r-1}\right) \quad , \quad k = 1, 2, \ldots, N_r, \tag{4.19}$$

the values of the Chebyshev polynomials at these points are simply given by the cosine functions:

$$\mathcal{C}_{nk} = \mathcal{C}_n(x_k) = \cos\left(\pi\frac{n(k-1)}{N_r-1}\right).$$

This particular choice has two advantages. For one, the grid points become denser toward the inner and outer radius and better resolve potential thermal and viscous boundary layers. In addition, FFTs can be employed to switch between grid representation (4.16) and Chebyshev representations (4.17), rendering this procedure a fast-Chebyshev transform. Choosing $N_r > N$ provides radial dealiasing.

See also:

The Chebyshev (Gauss-Lobatto) grid is defined in the module `chebyshev_polynoms_mod`. The cosine transforms are computed in the modules `cosine_transform_even` and `fft_fac_mod`.

4.4 Spectral equations

We have now introduced the necessary tools for deriving the spectral equations. Taking the **radial components** of the Navier-Stokes equation and the induction equation provides the equations for the poloidal potentials $W(r,\theta,\phi)$ and $g(r,\theta,\phi)$. The **radial component of the curl** of these equations provides the equations for the toroidal counterparts $Z(r,\theta,\phi)$ and $h(r,\theta,\phi)$. The pressure remains an additional unknown. Hence one more equation involving $W_{\ell mn}$ and $p_{\ell mn}$ is required. It is obtained by taking the **horizontal divergence** of the Navier-Stokes equation.

Expanding all potentials in spherical harmonics and Chebyshev polynomials, multiplying with $Y_\ell^{m\star}$, and integrating over spherical surfaces (while making use of the orthogonality relation (4.6) results in equations for the coefficients $W_{\ell mn}$, $Z_{\ell mn}$, $g_{\ell mn}$, $h_{\ell mn}$, $P_{\ell mn}$ and $s_{\ell mn}$, respectively.

4.4.1 Equation for the poloidal potential W

The temporal evolution of W is obtained by taking $\boldsymbol{e_r}\cdot$ of each term entering the Navier-Stokes equation. For the time-derivative, one gets using (4.2):

$$\tilde{\rho}\boldsymbol{e_r}\cdot\left(\frac{\partial \boldsymbol{u}}{\partial t}\right)=\frac{\partial}{\partial t}(\boldsymbol{e_r}\cdot\tilde{\rho}\boldsymbol{u})=-\Delta_H\frac{\partial W}{\partial t}.$$

For the viscosity term, one gets

$$\begin{aligned}\boldsymbol{e_r}\cdot\boldsymbol{\nabla}\cdot\mathsf{S}=&-\nu\,\Delta_H\left[\frac{\partial^2 W}{\partial r^2}+\left\{2\frac{d\ln\nu}{dr}-\frac{1}{3}\frac{d\ln\tilde{\rho}}{dr}\right\}\frac{\partial W}{\partial r}\right.\\&\left.-\left\{-\Delta_H+\frac{4}{3}\left(\frac{d^2\ln\tilde{\rho}}{dr^2}+\frac{d\ln\nu}{dr}\frac{d\ln\tilde{\rho}}{dr}+\frac{1}{r}\left[3\frac{d\ln\nu}{dr}+\frac{d\ln\tilde{\rho}}{dr}\right]\right)\right\}W\right],\end{aligned}$$

Note: In case of a constant kinematic viscosity, the $d\ln\nu/dr$ terms vanish. If in addition,the background density is constant, the $d\ln\tilde{\rho}/dr$ terms also vanish. In that Boussinesq limit, this viscosity term would then be simplified as

$$\boldsymbol{e_r}\cdot\Delta\boldsymbol{u}=-\Delta_H\left[\frac{\partial^2 W}{\partial r^2}+\Delta_H W\right].$$

Using Eq. (4.10) then allows to finally write the time-evolution equation for the poloidal potential $W_{\ell mn}$:

$$\begin{aligned}E\,\frac{\ell(\ell+1)}{r^2}\Bigg[&\left\{\frac{\partial}{\partial t}+\nu\,\frac{\ell(\ell+1)}{r^2}+\frac{4}{3}\nu\left(\frac{d^2\ln\tilde{\rho}}{dr^2}+\frac{d\ln\nu}{dr}\frac{d\ln\tilde{\rho}}{dr}+\frac{1}{r}\left[3\frac{d\ln\nu}{dr}+\frac{d\ln\tilde{\rho}}{dr}\right]\right)\right\}\mathcal{C}_n\\&-\nu\left\{2\frac{d\ln\nu}{dr}-\frac{1}{3}\frac{d\ln\tilde{\rho}}{dr}\right\}\mathcal{C}_n'\\&-\nu\,\mathcal{C}_n''\qquad\Bigg]\quad W_{\ell mn}\\&+\left[\mathcal{C}_n'-\frac{d\ln\tilde{\rho}}{dr}\mathcal{C}_n\right]\qquad P_{\ell mn}\\&-\left[\frac{Ra\,E}{Pr}\,\tilde{\rho}\,g(r)\right]\mathcal{C}_n\qquad s_{\ell mn}\\=\mathcal{N}_{\ell m}^W=&\int d\Omega\,Y_\ell^{m\star}\,\mathcal{N}^W=\int d\Omega\,Y_\ell^{m\star}\,\boldsymbol{e_r}\cdot\boldsymbol{F}\,.\end{aligned}\tag{4.20}$$

Here, $d\Omega$ is the spherical surface element. We use the summation convention for the Chebyshev index n. The radial derivatives of Chebyshev polynomials are denoted by primes.

See also:

The exact computation of the linear terms of (4.20) are coded in the subroutines `get_wpMat`

4.4.2 Equation for the toroidal potential Z

The temporal evolution of Z is obtained by taking the radial component of the curl of the Navier-Stokes equation (i.e. $\boldsymbol{e_r}\cdot\boldsymbol{\nabla}\times$). For the time derivative, one gets using (4.2):

$$\boldsymbol{e_r}\cdot\boldsymbol{\nabla}\times\left(\frac{\partial\tilde{\rho}\boldsymbol{u}}{\partial t}\right)=\frac{\partial}{\partial t}(\boldsymbol{e_r}\cdot\boldsymbol{\nabla}\times\tilde{\rho}\boldsymbol{u})=-\frac{\partial}{\partial t}(\Delta_H Z)=-\Delta_H\frac{\partial Z}{\partial t}\,.$$

The pressure gradient, one has

$$\boldsymbol{\nabla}\times\left[\tilde{\rho}\boldsymbol{\nabla}\left(\frac{p'}{\tilde{\rho}}\right)\right]=\boldsymbol{\nabla}\tilde{\rho}\times\boldsymbol{\nabla}\left(\frac{p'}{\tilde{\rho}}\right)+\underbrace{\tilde{\rho}\boldsymbol{\nabla}\times\left[\boldsymbol{\nabla}\left(\frac{p'}{\tilde{\rho}}\right)\right]}_{=0}\,.$$

This expression has no component along $\boldsymbol{e_r}$, as a consequence, there is no pressure gradient contribution here. The gravity term also vanishes as $\boldsymbol{\nabla}\times(\tilde{\rho}g(r)\boldsymbol{e_r})$ has no radial component.

$$\begin{aligned}\boldsymbol{e_r}\cdot\boldsymbol{\nabla}\times[\boldsymbol{\nabla}\cdot\mathsf{S}]=&-\nu\,\Delta_H\left[\frac{\partial^2 Z}{\partial r^2}+\left(\frac{d\ln\nu}{dr}-\frac{d\ln\tilde{\rho}}{dr}\right)\frac{\partial Z}{\partial r}\right.\\&\left.-\left(\frac{d\ln\nu}{dr}\frac{d\ln\tilde{\rho}}{dr}+\frac{2}{r}\frac{d\ln\nu}{dr}+\frac{d^2\ln\tilde{\rho}}{dr^2}+\frac{2}{r}\frac{d\ln\tilde{\rho}}{dr}-\Delta_H\right)Z\right].\end{aligned}$$

Note: Once again, this viscous term can be greatly simplified in the Boussinesq limit:

$$\boldsymbol{e_r}\cdot\boldsymbol{\nabla}\times(\Delta\boldsymbol{u})=-\Delta_H\left[\frac{\partial^2 Z}{\partial r^2}+\Delta_H\,Z\right]\,.$$

Using Eq. (4.10) then allows to finally write the time-evolution equation for the poloidal potential $Z_{\ell mn}$:

$$\begin{aligned}&E\,\frac{\ell(\ell+1)}{r^2}\left[\left\{\frac{\partial}{\partial t}+\nu\,\frac{\ell(\ell+1)}{r^2}+\nu\left(\frac{d\ln\nu}{dr}\frac{d\ln\tilde{\rho}}{dr}+\frac{2}{r}\frac{d\ln\nu}{dr}+\frac{d^2\ln\tilde{\rho}}{dr^2}+\frac{2}{r}\frac{d\ln\tilde{\rho}}{dr}\right)\right\}\mathcal{C}_n\right.\\&\qquad\qquad-\nu\left(\frac{d\ln\nu}{dr}-\frac{d\ln\tilde{\rho}}{dr}\right)\mathcal{C}_n'\\&\qquad\qquad\left.-\nu\,\mathcal{C}_n''\right]\quad Z_{\ell mn}\\&\qquad=\mathcal{N}^Z_{\ell m}=\int d\Omega\,Y_\ell^{m\star}\,\mathcal{N}^Z=\int d\Omega\,Y_\ell^{m\star}\,\boldsymbol{e_r}\cdot(\boldsymbol{\nabla}\times\boldsymbol{F})\,.\end{aligned}\tag{4.21}$$

See also:

The exact computation of the linear terms of (4.21) are coded in the subroutines `get_zMat`

4.4.3 Equation for pressure P

The evolution of equation for pressure is obtained by taking the horizontal divergence (i.e. $\boldsymbol{\nabla}_H\cdot$) of the Navier-Stokes equation. This operator is defined such that

$$\boldsymbol{\nabla}_H\cdot\boldsymbol{a}=r\sin\frac{\partial(\sin\theta\,a_\theta)}{\partial\theta}+r\sin\frac{\partial a_\phi}{\partial\phi}.$$

This relates to the total divergence via:

$$\boldsymbol{\nabla}\cdot\boldsymbol{a} = \frac{1}{r^2}\frac{\partial(r^2 a_r)}{\partial r} + \boldsymbol{\nabla}_H\cdot\boldsymbol{a}.$$

The time-derivative term is thus expressed by

$$\begin{aligned}
\boldsymbol{\nabla}_H\cdot\left(\tilde{\rho}\frac{\partial\boldsymbol{u}}{\partial t}\right) &= \frac{\partial}{\partial t}\left[\boldsymbol{\nabla}_H\cdot(\tilde{\rho}\boldsymbol{u})\right], \\
&= \frac{\partial}{\partial t}\left[\boldsymbol{\nabla}\cdot(\tilde{\rho}\boldsymbol{u}) - \frac{1}{r^2}\frac{\partial(r^2\tilde{\rho}u_r)}{\partial r}\right], \\
&= -\frac{\partial}{\partial t}\left[\frac{\partial(\tilde{\rho}u_r)}{\partial r} + \frac{2\tilde{\rho}u_r}{r}\right], \\
&= \frac{\partial}{\partial t}\left[\frac{\partial(\Delta_H W)}{\partial r} + \frac{2}{r}\Delta_H W\right], \\
&= \Delta_H\frac{\partial}{\partial t}\left(\frac{\partial W}{\partial r}\right).
\end{aligned}$$

We note that the gravity term vanishes since $\boldsymbol{\nabla}_H\cdot(\tilde{\rho}g(r)\boldsymbol{e_r}) = 0$. Concerning the pressure gradient, one has

$$-\boldsymbol{\nabla}_H\cdot\left[\tilde{\rho}\boldsymbol{\nabla}\left(\frac{p'}{\tilde{\rho}}\right)\right] = -\left\{\boldsymbol{\nabla}\cdot\left[\tilde{\rho}\boldsymbol{\nabla}\left(\frac{p'}{\tilde{\rho}}\right)\right] - \frac{1}{r^2}\frac{\partial}{\partial r}\left[r^2\tilde{\rho}\frac{\partial}{\partial r}\left(\frac{p'}{\tilde{\rho}}\right)\right]\right\} = -\Delta_H\, p'.$$

The viscosity term then reads

$$\begin{aligned}
\boldsymbol{\nabla}_H\cdot(\boldsymbol{\nabla}\cdot\mathsf{S}) =& \nu\,\Delta_H\left[\frac{\partial^3 W}{\partial r^3} + \left(\frac{d\ln\nu}{dr} - \frac{d\ln\tilde{\rho}}{dr}\right)\frac{\partial^2 W}{\partial r^2}\right. \\
&- \left[\frac{d^2\ln\tilde{\rho}}{dr^2} + \frac{d\ln\nu}{dr}\frac{d\ln\tilde{\rho}}{dr} + \frac{2}{r}\left(\frac{d\ln\nu}{dr} + \frac{d\ln\tilde{\rho}}{dr}\right) - \Delta_H\right]\frac{\partial W}{\partial r} \\
&\left. - \left(\frac{2}{3}\frac{d\ln\tilde{\rho}}{dr} + \frac{2}{r} + \frac{d\ln\nu}{dr}\right)\Delta_H\, W\right].
\end{aligned}$$

Note: Once again, this viscous term can be greatly simplified in the Boussinesq limit:

$$\boldsymbol{\nabla}_H\cdot(\Delta\boldsymbol{u}) = -\Delta_H\left[\frac{\partial^3 W}{\partial r^3} + \Delta_H\,\frac{\partial W}{\partial r} - \frac{2}{r}\Delta_H\, W\right].$$

Using Eq. (4.10) then allows to finally write the equation for the pressure $P_{\ell mn}$:

$$\begin{aligned}
& E\,\frac{\ell(\ell+1)}{r^2}\left[-\nu\left(\frac{2}{3}\frac{d\ln\tilde{\rho}}{dr} + \frac{2}{r} + \frac{d\ln\nu}{dr}\right)\frac{\ell(\ell+1)}{r^2}\,\mathcal{C}_n\right. \\
&\left\{\frac{\partial}{\partial t} + \nu\,\frac{\ell(\ell+1)}{r^2} + \nu\left[\frac{d^2\ln\tilde{\rho}}{dr^2} + \frac{d\ln\nu}{dr}\frac{d\ln\tilde{\rho}}{dr} + \frac{2}{r}\left(\frac{d\ln\nu}{dr} + \frac{d\ln\tilde{\rho}}{dr}\right)\right]\right\}\mathcal{C}'_n \\
&\qquad -\nu\left(\frac{d\ln\nu}{dr} - \frac{d\ln\tilde{\rho}}{dr}\right)\mathcal{C}''_n \\
&\qquad\qquad -\nu\,\mathcal{C}'''_n \qquad\qquad \Big] \quad W_{\ell mn} \\
&\qquad\qquad + \left[\frac{\ell(\ell+1)}{r^2}\right]\mathcal{C}_n \qquad\qquad P_{\ell mn} \\
&= \mathcal{N}^P_{\ell m} = -\int d\Omega\, Y_\ell^{m\star}\,\mathcal{N}^P = -\int d\Omega\, Y_\ell^{m\star}\,\boldsymbol{\nabla}_H\cdot\boldsymbol{F}\,.
\end{aligned} \tag{4.22}$$

See also:

The exact computation of the linear terms of (4.22) are coded in the subroutines `get_wpMat`

Note: We note that the terms on the left hand side of (4.20), (4.21) and (4.22) resulting from the viscous term, the pressure gradient, the buoyancy term, and the explicit time derivative completely decouple in spherical harmonic degree and order.

The terms that do not decouple, namely Coriolis force, Lorentz force and advection of momentum, are collected on the right-hand side of (4.20), (4.21) and (4.22) into the forcing term $\boldsymbol{F}$:

$$\boldsymbol{F} = -2\,\tilde{\rho}\,\boldsymbol{e_z}\times\boldsymbol{u} - E\,\tilde{\rho}\,\boldsymbol{u}\cdot\boldsymbol{\nabla}\,\boldsymbol{u} + \frac{1}{Pm}\,(\boldsymbol{\nabla}\times\boldsymbol{B})\times\boldsymbol{B}\,. \tag{4.23}$$

Resolving $\boldsymbol{F}$ into potential functions is not required. Its numerical evaluation is discussed *below*.

4.4.4 Equation for entropy s

The equation for the entropy (or temperature in the Boussinesq limit) is given by

$$\begin{aligned}
&\frac{1}{Pr}\left[\left(Pr\frac{\partial}{\partial t}+\kappa\,\frac{\ell(\ell+1)}{r^2}\right)\mathcal{C}_n\right.\\
&\quad -\kappa\left(\frac{d\ln\kappa}{dr}+\frac{d\ln\tilde{\rho}}{dr}++\frac{dln\tilde{T}}{dr}+\frac{2}{r}\right)\mathcal{C}_n'\\
&\quad \left.-\kappa\,\mathcal{C}_n''\right]\; s_{\ell mn}\\
&=\mathcal{N}^S_{\ell m}=\int d\Omega\,Y_\ell^{m\star}\,\mathcal{N}^S=\int d\Omega\,Y_\ell^{m\star}\left[-\boldsymbol{u}\cdot\boldsymbol{\nabla}s+\frac{Pr\,Di}{Ra}\frac{1}{\tilde{\rho}\tilde{T}}\left(\Phi_\nu+\frac{\lambda}{Pm^2\,E}\,j^2\right)\right]\,.
\end{aligned} \tag{4.24}$$

In this expression, $j=\boldsymbol{\nabla}\times\boldsymbol{B}$ is the current. Once again, the numerical evaluation of the right-hand-side (i.e. the non-linear terms) is discussed *below*.

See also:

The exact computation of the linear terms of (4.24) are coded in the subroutines `get_sMat`

4.4.5 Equation for the poloidal magnetic potential g

The equation for the poloidal magnetic potential is the radial component of the dynamo equation since

$$\boldsymbol{e_r}\cdot\left(\frac{\partial\boldsymbol{B}}{\partial t}\right)=\frac{\partial}{\partial t}(\boldsymbol{e_r}\cdot\boldsymbol{B})=-\Delta_H\frac{\partial g}{\partial t}.$$

The spectral form then reads

$$\begin{aligned}
&\frac{\ell(\ell+1)}{r^2}\left[\left(\frac{\partial}{\partial t}+\frac{1}{Pm}\lambda\,\frac{\ell(\ell+1)}{r^2}\right)\mathcal{C}_n\right.\\
&\quad\left.-\frac{1}{Pm}\,\lambda\,\mathcal{C}_n''\right]\; g_{\ell mn}\\
&=\mathcal{N}^g_{\ell m}=\int d\Omega\,Y_\ell^{m\star}\,\mathcal{N}^g=\int d\Omega\,Y_\ell^{m\star}\,\boldsymbol{e_r}\cdot\boldsymbol{D}\,.
\end{aligned} \tag{4.25}$$

See also:

The exact computation of the linear terms of (4.25) are coded in the subroutines `get_bMat`

4.4.6 Equation for the toroidal magnetic potential h

The equation for the toroidal magnetic field coefficient reads

$$\begin{aligned} &\frac{\ell(\ell+1)}{r^2}\left[\left(\frac{\partial}{\partial t}+\frac{1}{Pm}\lambda\frac{\ell(\ell+1)}{r^2}\right)\mathcal{C}_n\right. \\ &\qquad -\frac{1}{Pm}\frac{d\lambda}{dr}\mathcal{C}'_n \\ &\qquad \left.-\frac{1}{Pm}\lambda\mathcal{C}''_n\right] \quad h_{\ell mn} \\ &= \mathcal{N}^h_{\ell m} = \int d\Omega\, Y_\ell^{m\star}\,\mathcal{N}^h = \int d\Omega\, Y_\ell^{m\star}\,\boldsymbol{e_r}\cdot(\boldsymbol{\nabla}\times\boldsymbol{D})\ . \end{aligned} \tag{4.26}$$

See also:

The exact computation of the linear terms of (4.26) are coded in the subroutines `get_bMat`

Note: We note that the terms on the left hand side of (4.25) and (4.26) resulting from the magnetic diffusion term and the explicit time derivative completely decouple in spherical harmonic degree and order.

The dynamo term does not decouple:

$$\boldsymbol{D} = \boldsymbol{\nabla}\times(\boldsymbol{u}\times\boldsymbol{B})\ . \tag{4.27}$$

We have now derived a full set of equations (4.20), (4.21), (4.22), (4.24), (4.25) and (4.26), each describing the evolution of a single spherical harmonic mode of the six unknown fields (assuming that the terms on the right hand side are given). Each equation couples $N+1$ Chebyshev coefficients for a given spherical harmonic mode (ℓ, m). Typically, a collocation method is employed to solve for the Chebyshev coefficients. This means that the equations are required to be exactly satisfied at $N-1$ grid points defined by the equations (4.18) and (4.19). Excluded are the points $r = r_i$ and $r = r_o$, where the *boundary conditions* provide additional constraints on the set of Chebyshev coefficients.

4.5 Time-stepping schemes

Implicit time stepping schemes theoretically offer increased stability and allow for larger time steps. However, fully implicit approaches have the disadvantage that the nonlinear-terms couple all spherical harmonic modes. The potential gain in computational speed is therefore lost at higher resolution, where one very large matrix has to be dealt with rather than a set of much smaller ones. Similar considerations hold for the Coriolis force, one of the dominating forces in the system and therefore a prime candidate for implicit treatment. However, the Coriolis term couples modes (ℓ, m, n) with $(\ell+1, m, n)$ and $(\ell-1, m, n)$ and also couples poloidal and toroidal flow potentials. An implicit treatment of the Coriolis term therefore also results in a much larger (albeit sparse) inversion matrix.

We consequently adopt in **MagIC** a mixed implicit/explicit algorithm. The general differential equation in time can be written in the form

$$\frac{\partial}{\partial t}x + \mathcal{I}(x,t) = \mathcal{E}(x,t)\ .$$

where $\mathcal{I}$ denotes the terms treated in an implicit time step and $\mathcal{E}$ the terms treated explicitly, i.e.~the nonlinear and Coriolis contributions. The discretized implicit time step is given by

$$\left(\frac{x(t+\delta t)-x(t)}{\delta t}\right)_I = -\alpha\,\mathcal{I}(x,t+\delta t) - (1-\alpha)\,\mathcal{I}(x,t)\ .$$

where α is the weight of the new time step. For $\alpha = 0.5$ we recover a classic Crank-Nicolson algorithm.

A second order Adams-Bashforth scheme is used for the explicit time step:

$$\left(\frac{x(t+\delta t)-x(t)}{\delta t}\right)_E = \frac{3}{2}\,\mathcal{E}(x,t) - \frac{1}{2}\,\mathcal{E}(x,t-\delta t) \ .$$

The combination of both steps yields

$$\frac{x(t+\delta t)}{\delta t} + \alpha\,\mathcal{I}(x,t+\delta t) = \frac{x(t)}{\delta t} - (1-\alpha)\,\mathcal{I}(x,t) + \frac{3}{2}\,\mathcal{E}(x,t) - \frac{1}{2}\,\mathcal{E}(x,t-\delta t) \ .$$

In the code such an equation is formulated for each unknown spectral coefficient (expect pressure) of spherical harmonic degree ℓ and order m and for each radial grid point r_k. Because non-linear terms and the Coriolis force are treated explicitly, the equations decouple for all spherical modes. The different radial grid points, however, couple via the Chebychev modes and form a linear algebraic system of equations that can be solved with standard methods for the different spectral contributions.

For example the respective system of equations for the modes of the poloidal magnetic potential g results from considering the radial component of the dynamo equation:

$$(\mathcal{A}_{kn} + \alpha\,\mathcal{I}_{kn})\ g_{\ell mn}(t+\delta t) = (\mathcal{A}_{kn} - (1-\alpha)\,\mathcal{I}_{kn})\ g_{\ell mn}(t) + \frac{3}{2}\,\mathcal{E}_{k\ell m}(t) - \frac{1}{2}\,\mathcal{E}_{k\ell m}(t-\delta t) \tag{4.28}$$

with

$$\mathcal{A}_{kn} = \frac{\ell(\ell+1)}{r_k^2}\,\frac{1}{\delta t}\mathcal{C}_{nk} \ ,$$

$$\mathcal{I}_{kn} = \frac{\ell(\ell+1)}{r_k^2}\,\frac{1}{Pm}\left(\frac{\ell(\ell+1)}{r_k^2}\,\mathcal{C}_{nk} - \mathcal{C}''_{nk}\right) \ ,$$

and $\mathcal{C}_{nk} = \mathcal{C}_n(r_k)$. $\mathcal{A}_{kn}$ is a matrix that converts the poloidal field modes $g_{\ell mn}$ to the radial magnetic field $B_r(r_k,\ell m)$ for a given spherical harmonic contribution with an additional division by the time step δt:

Here k and n number the radial grid points and the Chebychev coefficients, respectively. Note that the Einstein sum convention is used for Chebychev modes n.

$\mathcal{I}_{kn}$ is the matrix describing the implicit contribution which is purely diffusive here. Neither $\mathcal{A}_{kn}$ nor $\mathcal{I}_{kn}$ depend on time but the former needs to be updated when the time step δt is changed. The only explicit contribution is the nonlinear dynamo term

$$\mathcal{E}_{k\ell m}(t) = \mathcal{N}^g_{k\ell m} = \int d\Omega\ Y_\ell^{m\star}\ \boldsymbol{e_r}\cdot\boldsymbol{D}(t,r_k,\theta,\phi) \ .$$

$\mathcal{E}_{k\ell m}$ is a one dimensional vector for all spherical harmonic combinations ℓm.

How are these operations organized in the code? Within MagIC the poloidal magnetic field potential is called `b`. The implicit step for the current time t and the explicit step for the previous time $t-\delta t$ is combined into one two-dimensional array

$$\texttt{dbdtLast}(k,lm) = -\frac{1}{2}\,\mathcal{E}_{k\ell m}(t-\delta t) - (1-\alpha)\,\mathcal{I}_{kn}\,g_{\ell mn}(t) \ .$$

where k numbers the first dimension and the second dimension lm numbers the spherical harmonic modes. The explicit time step part is called `dbdt` in MagIC:

$$\texttt{dbdt}(k,\ell m) = \mathcal{E}_{k\ell m}.$$

The combination of left hand side of (4.28) provides the time stepping matrix for g:

$$\texttt{bmat}(k,n,\ell) = A_{kn} + \alpha\mathcal{I}_{kn} \ .$$

There is a different time stepping matrix for each spherical harmonic degree ℓ. The linear system of equations solved for the mode $\mathtt{b}(k,\ell m)$ at time $t = t + \delta t$ is then

$$\begin{aligned}\mathtt{bmat}(k,n,\ell) \star \mathtt{b}_{i+1}(n,\ell m) =& \mathtt{w1} \star \mathtt{dbdt}(k,\ell m) + \mathtt{w2} \star \mathtt{dbdtLast}(k,\ell m) + \\ & \mathtt{Odt} \star \mathtt{dLh}(\ell) \star \mathtt{Or2}(k) \star \mathtt{b}_i(n,\ell m) \\ =& \mathtt{rhs}(k,\ell m)\end{aligned}$$

with

$$\mathtt{w1} = -1/2\frac{\delta t}{\delta t_{old}} \ , \ \mathtt{w2} = 1 - \mathtt{w2} \ , \ \mathtt{Odt} = 1/\delta t \ , \ \mathtt{dLh}(\ell) = \ell(\ell+1) \ , \ \mathtt{Or2}(k) = 1/r_k^2.$$

The respective equations for the poloidal flow potential are somewhat more complex and involve coupling to the pressure.....

Note: The poloidal flow potential (4.20) and the pressure (4.22) are nevertheless coupled for a given spherical harmonic mode. Likewise the poloidal flow and entropy (or temperature) equations should also couple, but here MagIC takes the shortcut of updating entropy first and using the already updated value for an explicit treatment of buoyancy.

Courant's condition offers a guideline concerning the value of δt, demanding that δt should be smaller than the advection time between two grid points. Strong Lorentz forces require an additional stability criterion that is obtained by replacing the flow speed by Alfvén's velocity in a modified Courant criterion. The explicit treatment of the Coriolis force requires that the time step is limited to a fraction of the rotation period, which may be the relevant criterion at low Ekman number when flow and magnetic field remain weak. Non-homogeneous grids and other numerical effects generally require an additional safety factor in the choice of δt.

4.6 Coriolis force and nonlinear terms

4.6.1 Nonlinear terms entering the equation for W

The nonlinear term $\mathcal{N}^W$ that enters the equation for the poloidal potential (4.20) contains the radial component of the advection, the Lorentz force and Coriolis force. In spherical coordinate, the two first contributions read:

$$\tilde{\rho}\left(\boldsymbol{u}\cdot\boldsymbol{\nabla}\boldsymbol{u}\right) = \left\{\begin{array}{c}\mathcal{A}_r\\ \mathcal{A}_\theta\\ \mathcal{A}_\phi\end{array}\right\} = \left\{\begin{array}{r} -\tilde{\rho}\,E\left(u_r\dfrac{\partial u_r}{\partial r}+\dfrac{u_\theta}{r}\dfrac{\partial u_r}{\partial\theta}+\dfrac{u_\phi}{r\sin\theta}\dfrac{\partial u_r}{\partial\phi}-\dfrac{u_\theta^2+u_\phi^2}{r}\right)+\dfrac{1}{Pm}\left(j_\theta\,B_\phi-j_\phi\,B_\theta\right)\ ,\\ -\tilde{\rho}\,E\left(u_r\dfrac{\partial u_\theta}{\partial r}+\dfrac{u_\theta}{r}\dfrac{\partial u_\theta}{\partial\theta}+\dfrac{u_\phi}{r\sin\theta}\dfrac{\partial u_\theta}{\partial\phi}+\dfrac{u_r u_\theta}{r}-\dfrac{\cos\theta}{r\sin\theta}u_\phi^2\right)+\dfrac{1}{Pm}\left(j_\phi\,B_r-j_r\,B_\phi\right)\ ,\\ -\tilde{\rho}\,E\left(u_r\dfrac{\partial u_\phi}{\partial r}+\dfrac{u_\theta}{r}\dfrac{\partial u_\phi}{\partial\theta}+\dfrac{u_\phi}{r\sin\theta}\dfrac{\partial u_\phi}{\partial\phi}+\dfrac{u_r u_\phi}{r}+\dfrac{\cos\theta}{r\sin\theta}u_\theta u_\phi\right)+\dfrac{1}{Pm}\left(j_r\,B_\theta-j_\theta\,B_r\right)\ ,\end{array}\right\} \tag{4.29}$$

The Coriolis force can be expressed as a function of the potentials W and Z using (4.4)

$$2\tilde{\rho}\boldsymbol{e_r}\cdot\left(\boldsymbol{u}\times\boldsymbol{e_z}\right) = 2\sin\theta\,\tilde{\rho}u_\phi = \frac{2}{r}\left(\frac{\partial^2 W}{\partial r\partial\phi}-\sin\theta\frac{\partial Z}{\partial\theta}\right)\ .$$

The nonlinear terms that enter the equation for the poloidal potential (4.20) thus reads:

$$\mathcal{N}^W = \frac{2}{r}\left(\frac{\partial^2 W}{\partial r\partial\phi}-\sin\theta\frac{\partial Z}{\partial\theta}\right)+\mathcal{A}_r\,.$$

The θ-derivative entering the radial component of the Coriolis force is thus the operator ϑ_2 defined in (4.12). Using the recurrence relation, one thus finally gets in spherical harmonic space:

$$\boxed{\mathcal{N}_{\ell m}^{W} = \frac{2}{r}\left[im\frac{\partial W_\ell^m}{\partial r} - (\ell-1)c_\ell^m Z_{\ell-1}^m + (\ell+2)c_{\ell+1}^m Z_{\ell+1}^m\right] + \mathcal{A}_{r\ell}{}^m .} \tag{4.30}$$

To get this expression, we need to first compute $\mathcal{A}_r$ in the physical space. This term is computed in the subroutine `get_nl` in the module `grid_space_arrays_mod`. $\mathcal{A}_r$ is then transformed to the spectral space by using a Legendre and a Fourier transform to produce $\mathcal{A}_{r\ell}{}^m$.

See also:

The final calculations of (4.30) are done in the subroutine *get_td*.

4.6.2 Nonlinear terms entering the equation for Z

The nonlinear term $\mathcal{N}^Z$ that enters the equation for the toroidal potential (4.21) contains the radial component of the curl of the advection and Coriolis force. The Coriolis force can be rewritten as a function of W and Z:

$$\begin{aligned}
\boldsymbol{e_r}\cdot\boldsymbol{\nabla}\times[(2\tilde{\rho}\boldsymbol{u})\times\boldsymbol{e_z}] &= 2\boldsymbol{e_r}\cdot[(\boldsymbol{e_z}\cdot\boldsymbol{\nabla})(\tilde{\rho}\boldsymbol{u})]\,,\\
&= 2\left[\cos\theta\frac{\partial(\tilde{\rho}u_r)}{\partial r} - \frac{\sin\theta}{r}\frac{\partial(\tilde{\rho}u_r)}{\partial\theta} + \frac{\tilde{\rho}u_\theta\sin\theta}{r}\right],\\
&= 2\left[-\cos\theta\frac{\partial}{\partial r}(\Delta_H W) + \frac{\sin\theta}{r}\frac{\partial}{\partial\theta}(\Delta_H W) + \frac{\sin\theta}{r^2}\frac{\partial^2 W}{\partial r\partial\theta} + \frac{1}{r^2}\frac{\partial Z}{\partial\phi}\right].
\end{aligned}$$

Using the ϑ operators defined in (4.12)-(4.15) then allows to rewrite the Coriolis force in the following way:

$$\boldsymbol{e_r}\cdot\boldsymbol{\nabla}\times[(2\tilde{\rho}\boldsymbol{u})\times\boldsymbol{e_z}] = \frac{2}{r^2}\left(\vartheta_3\,\frac{\partial W}{\partial r} - \frac{1}{r}\,\vartheta_4\,W + \frac{\partial Z}{\partial\phi}\right). \tag{4.31}$$

The contributions of nonlinear advection and Lorentz forces that enter the equation for the toroidal potential are written this way:

$$\frac{1}{r\sin\theta}\left[\frac{\partial(\sin\theta\mathcal{A}_\phi)}{\partial\theta} - \frac{\partial\mathcal{A}_\theta}{\partial\phi}\right].$$

To make use of the recurrence relations (4.12)-(4.15), the actual strategy is to follow the following steps:

1. Compute the quantities $\mathcal{A}_\phi/r\sin\theta$ and $\mathcal{A}_\theta/r\sin\theta$ in the physical space. In the code, this step is computed in the subroutine `get_nl` in the module `grid_space_arrays_mod`.
2. Transform $\mathcal{A}_\phi/r\sin\theta$ and $\mathcal{A}_\theta/r\sin\theta$ to the spectral space (thanks to a Legendre and a Fourier transform). In MagIC, this step is computed in the modules *legendre_grid_to_spec* and *fft*. After this step $\mathcal{A}t_\ell^m$ and $\mathcal{A}p_\ell^m$ are defined.
3. Calculate the colatitude and theta derivatives using the recurrence relations:

$$\vartheta_1\,\mathcal{A}p_\ell^m - \frac{\partial\mathcal{A}t_\ell^m}{\partial\phi}\,. \tag{4.32}$$

Using (4.31) and (4.32), one thus finally gets

$$\begin{aligned}
\boxed{\mathcal{N}_{\ell m}^{Z} =& \frac{2}{r^2}\left[(\ell-1)(\ell+1)\,c_\ell^m\,\frac{\partial W_{\ell-1}^m}{\partial r} + \ell(\ell+2)\,c_{\ell+1}^m\,\frac{\partial W_{\ell+1}^m}{\partial r}\right.}\\
&\left.-\frac{\ell(\ell-1)(\ell+1)}{r}\,c_\ell^m\,W_{\ell-1}^m + \frac{\ell(\ell+1)(\ell+2)}{r}\,c_{\ell+1}^m\,W_{\ell+1}^m + im\,Z_\ell^m\right]\\
&+(\ell+1)\,c_\ell^m\,\mathcal{A}p_{\ell-1}^m - \ell\,c_{\ell+1}^m\,\mathcal{A}p_{\ell+1}^m - im\,\mathcal{A}t_\ell^m\,.
\end{aligned} \tag{4.33}$$

See also:

The final calculations of (4.33) are done in the subroutine `get_td`.

4.6.3 Nonlinear terms entering the equation for P

The nonlinear term $\mathcal{N}^P$ that enters the equation for the pressure (4.22) contains the horizontal divergence of the advection and Coriolis force. The Coriolis force can be rewritten as a function of W and Z:

$$\begin{aligned}
\boldsymbol{\nabla}_H \cdot [(2\tilde{\rho}\boldsymbol{u}) \times \boldsymbol{e_z}] &= 2\boldsymbol{e_z} \cdot [\boldsymbol{\nabla} \times (\tilde{\rho}\boldsymbol{u})] - \left(\frac{\partial}{\partial r} + \frac{2}{r}\right) [\boldsymbol{e_r} \cdot (2\tilde{\rho}\boldsymbol{u} \times \boldsymbol{e_z})] \,, \\
&= -2\cos\theta\, \Delta_H Z - 2\sin\theta \left[-\frac{1}{r\sin\theta} \frac{\partial}{\partial\phi} \left(\frac{\partial^2}{\partial r^2} + \Delta_H\right) W + \frac{1}{r} \frac{\partial^2 Z}{\partial r \partial\theta} \right] \\
&\quad - \left(\frac{\partial}{\partial r} + \frac{2}{r}\right) [2\sin\theta \tilde{\rho} u_\phi] \,, \\
&= 2\left[\frac{1}{r} \left(\Delta_H + \frac{\partial^2}{\partial r^2}\right) \frac{\partial W}{\partial \phi} - \cos\theta \Delta_H Z - \frac{\sin\theta}{r} \frac{\partial^2 Z}{\partial r \partial\theta} \right] \\
&\quad - \left(\frac{\partial}{\partial r} + \frac{2}{r}\right) \left[\frac{2}{r} \left(\frac{\partial^2 W}{\partial r \partial\phi} - \sin\theta \frac{\partial Z}{\partial\theta} \right) \right] , \\
&= 2\left(\frac{\Delta_H}{r} \frac{\partial W}{\partial\phi} - \frac{1}{r^2} \frac{\partial^2 W}{\partial\phi \partial r} - \cos\theta \Delta_H\, Z + \frac{\sin\theta}{r^2} \frac{\partial Z}{\partial\theta} \right) .
\end{aligned}$$

Using the ϑ operators defined in (4.14)-(4.15) then allows to rewrite the Coriolis force in the following way:

$$\boldsymbol{\nabla}_H \cdot [(2\tilde{\rho}\boldsymbol{u}) \times \boldsymbol{e_z}] = \frac{2}{r^2} \left(-\frac{L_H}{r} \frac{\partial W}{\partial\phi} - \frac{\partial^2 W}{\partial\phi \partial r} + \vartheta_3\, Z \right) . \tag{4.34}$$

The contributions of nonlinear advection and Lorentz forces that enter the equation for pressure are written this way:

$$\frac{1}{r\sin\theta} \left[\frac{\partial(\sin\theta \mathcal{A}_\theta)}{\partial\theta} + \frac{\partial \mathcal{A}_\phi}{\partial\phi} \right] .$$

To make use of the recurrence relations (4.12)-(4.15), we then follow the same three steps as for the advection term entering the equation for Z.

$$\vartheta_1\, \mathcal{A}t_\ell^m + \frac{\partial \mathcal{A}p_\ell^m}{\partial\phi} \,. \tag{4.35}$$

Using (4.34) and (4.35), one thus finally gets

$$\boxed{\begin{aligned}
\mathcal{N}_{\ell m}^P =& \frac{2}{r^2} \left[-im \frac{\ell(\ell+1)}{r} W_\ell^m - im \frac{\partial W_\ell^m}{\partial r} + (\ell-1)(\ell+1)\, c_\ell^m\, Z_{\ell-1}^m + \ell(\ell+2)\, c_{\ell+1}^m\, Z_{\ell+1}^m \right] \\
&+ (\ell+1)\, c_\ell^m\, \mathcal{A}t_{\ell-1}^m - \ell\, c_{\ell+1}^m\, \mathcal{A}t_{\ell+1}^m + im\, \mathcal{A}p_\ell^m \,.
\end{aligned}} \tag{4.36}$$

See also:

The final calculations of (4.36) are done in the subroutine `get_td`.

4.6.4 Nonlinear terms entering the equation for s

The nonlinear terms that enter the equation for entropy/temperature (4.24) are twofold: (i) the advection term, (ii) the viscous and Ohmic heating terms (that vanish in the Boussinesq limit of the Navier Stokes equations).

Viscous and Ohmic heating are directly calculated in the physical space by the subroutine `get_nl` in the module `grid_space_arrays_mod`. Let's introduce $\mathcal{H}$, the sum of the viscous and Ohmic heating terms.

$$\mathcal{H} = \frac{Pr\,Di}{Ra}\frac{1}{\tilde{\rho}\tilde{T}}\left(\Phi_\nu + \frac{\lambda}{Pm^2\,E}\,j^2\right) .$$

Expanding this term leads to:

$$\begin{aligned}\mathcal{H} =& \frac{Pr\,Di}{Ra}\frac{1}{\tilde{\rho}\tilde{T}}\left[\tilde{\rho}\nu\left\{2\left(\frac{\partial u_r}{\partial r}\right)^2+2\left(\frac{1}{r}\frac{\partial u_\theta}{\partial\theta}+\frac{u_r}{r}\right)^2+2\left(\frac{1}{r\sin\theta}\frac{\partial u_\phi}{\partial\phi}+\frac{u_r}{r}+\frac{\cos\theta}{r\sin\theta}u_\theta\right)^2\right.\right.\\ &+\left(r\frac{\partial}{\partial r}\left(\frac{u_\theta}{r}\right)+\frac{1}{r}\frac{\partial u_r}{\partial\theta}\right)^2+\left(r\frac{\partial}{\partial r}\left(\frac{u_\phi}{r}\right)+\frac{1}{r\sin\theta}\frac{\partial u_r}{\partial\phi}\right)^2\\ &\left.\left.+\left(\frac{\sin\theta}{r}\frac{\partial}{\partial\theta}\left(\frac{u_\phi}{\sin\theta}\right)+\frac{1}{r\sin\theta}\frac{\partial u_\theta}{\partial\phi}\right)^2-\frac{2}{3}\left(\frac{d\ln\tilde{\rho}}{dr}u_r\right)^2\right\}\right.\\ &\left.+\frac{\lambda}{Pm^2\,E}\left\{j_r^2+j_\theta^2+j_\phi^2\right\}\right] .\end{aligned} \tag{4.37}$$

This term is then transformed to the spectral space with a Legendre and a Fourier transform to produce $\mathcal{H}_\ell^m$.

The treatment of the advection term $-\boldsymbol{u}\cdot\boldsymbol{\nabla}s$ is a bit different. It is in a first step rearranged as follows

$$-\boldsymbol{u}\cdot\boldsymbol{\nabla}s = -\frac{1}{\tilde{\rho}}\left[\boldsymbol{\nabla}\cdot(\tilde{\rho}s\boldsymbol{u}) - \underbrace{\boldsymbol{\nabla}\cdot(\tilde{\rho}\boldsymbol{u})}_{=0}\right] .$$

The quantities that are calculated in the physical space are thus simply the product of entropy/temperature s by the velocity components. This defines three variables defined in the grid space that are computed in the subroutine `get_nl`:

$$\mathcal{US}_r = \tilde{\rho}su_r, \quad \mathcal{US}_\theta = \tilde{\rho}su_\theta, \quad \mathcal{US}_\phi = \tilde{\rho}su_\phi,$$

To get the actual advection term, one must then apply the divergence operator to get:

$$-\boldsymbol{u}\cdot\boldsymbol{\nabla}s = -\frac{1}{\tilde{\rho}}\left[\frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\mathcal{US}_r\right)+\frac{1}{r\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\mathcal{US}_\theta\right)+\frac{1}{r\sin\theta}\frac{\partial\mathcal{US}_\phi}{\partial\phi}\right] .$$

To make use of the recurrence relations (4.12)-(4.15), the actual strategy is then to follow the following steps:

1. Compute the quantities $r^2\mathcal{US}_r$, $\mathcal{US}_\phi/r\sin\theta$ and $\mathcal{US}_\theta/r\sin\theta$ in the physical space. In the code, this step is computed in the subroutine `get_nl` in the module `grid_space_arrays_mod`.
2. Transform $r^2\mathcal{US}_r$, $\mathcal{US}_\phi/r\sin\theta$ and $\mathcal{US}_\theta/r\sin\theta$ to the spectral space (thanks to a Legendre and a Fourier transform). In MagIC, this step is computed in the modules *`legendre_grid_to_spec`* and *`fft`*. After this step $\mathcal{USr}_\ell^m$, $\mathcal{USt}_\ell^m$ and $\mathcal{USp}_\ell^m$ are defined.
3. Calculate the colatitude and theta derivatives using the recurrence relations:

$$-\frac{1}{\tilde{\rho}}\left[\frac{1}{r^2}\frac{\partial\mathcal{USr}_\ell^m}{\partial r}+\vartheta_1\mathcal{USt}_\ell^m+\frac{\partial\mathcal{USp}_\ell^m}{\partial\phi}\right] . \tag{4.38}$$

Using (4.37) and (4.38), one thus finally gets

$$\boxed{\mathcal{N}_{\ell m}^S = -\frac{1}{\tilde{\rho}}\left[\frac{1}{r^2}\frac{\partial\mathcal{USr}_\ell^m}{\partial r}+(\ell+1)\,c_\ell^m\,\mathcal{USt}_{\ell-1}^m-\ell\,c_{\ell+1}^m\mathcal{USt}_{\ell+1}^m+im\mathcal{USp}_\ell^m\right]+\mathcal{H}_\ell^m .} \tag{4.39}$$

See also:

The θ and ϕ derivatives that enter (4.39) are done in the subroutine *`get_td`*. The radial derivative is computed afterwards at the very beginning of *`updateS`*.

4.6.5 Nonlinear terms entering the equation for g

The nonlinear term that enters the equation for the poloidal potential of the magnetic field (4.25) is the radial component of the induction term (4.27). In the following we introduce the electromotive force $\mathcal{F} = \boldsymbol{u} \times \boldsymbol{B}$ with its three components

$$\mathcal{F}_r = u_\theta B_\phi - u_\phi B_\theta, \quad \mathcal{F}_\theta = u_\phi B_r - u_r B_\phi, \quad \mathcal{F}_\phi = u_r B_\theta - u_\theta B_r \,.$$

The radial component of the induction term then reads:

$$\mathcal{N}^g = \boldsymbol{e_r} \cdot [\boldsymbol{\nabla} \times (\boldsymbol{u} \times \boldsymbol{B})] = \frac{1}{r \sin\theta} \left[\frac{\partial \left(\sin\theta \mathcal{F}_\phi\right)}{\partial \theta} - \frac{\partial \mathcal{F}_\theta}{\partial \phi} \right] .$$

To make use of the recurrence relations (4.12)-(4.15), we then follow the usual following steps:

1. Compute the quantities $r^2\,\mathcal{F}_r$, $\mathcal{F}_\phi / r\sin\theta$ and $\mathcal{F}_\theta / r\sin\theta$ in the physical space. In the code, this step is computed in the subroutine `get_nl` in the module `grid_space_arrays_mod`.
2. Transform $r^2\,\mathcal{F}_r$, $\mathcal{F}_\phi / r\sin\theta$ and $\mathcal{F}_\theta / r\sin\theta$ to the spectral space (thanks to a Legendre and a Fourier transform). In MagIC, this step is computed in the modules *`legendre_grid_to_spec`* and *`fft`*. After this step ${\mathcal{F}_r}_\ell^m$, ${\mathcal{F}_\theta}_\ell^m$ and ${\mathcal{F}_\phi}_\ell^m$ are defined.
3. Calculate the colatitude and theta derivatives using the recurrence relations:

$$\vartheta_1 \, {\mathcal{F}_\phi}_\ell^m - \frac{\partial \, {\mathcal{F}_\theta}_\ell^m}{\partial \phi} \,.$$

We thus finally get

$$\boxed{\mathcal{N}^g_{\ell m} = (\ell+1)\, c_\ell^m \, {\mathcal{F}_\phi}_{\ell-1}^m - \ell \, c_{\ell+1}^m \, {\mathcal{F}_\phi}_{\ell+1}^m - im \, {\mathcal{F}_\theta}_\ell^m \,.} \tag{4.40}$$

See also:

The final calculations of (4.40) are done in the subroutine *`get_td`*.

4.6.6 Nonlinear terms entering the equation for h

The nonlinear term that enters the equation for the toroidal potential of the magnetic field (4.26) is the radial component of the curl of the induction term (4.27):

$$\begin{aligned}
\mathcal{N}^h = \boldsymbol{e_r} \cdot [\boldsymbol{\nabla} \times \boldsymbol{\nabla} \times (\boldsymbol{u} \times \boldsymbol{B})] &= \boldsymbol{e_r} \cdot [\boldsymbol{\nabla} (\boldsymbol{\nabla} \cdot \boldsymbol{\mathcal{F}}) - \Delta \boldsymbol{\mathcal{F}}] \,, \\
&= \frac{\partial}{\partial r} \left[\frac{1}{r^2} \frac{\partial (r^2 \mathcal{F}_r)}{\partial r} + \frac{1}{r\sin\theta} \frac{\partial (\sin\theta \, \mathcal{F}_\theta)}{\partial \theta} + \frac{1}{r\sin\theta} \frac{\partial \mathcal{F}_\phi}{\partial \phi} \right] \\
&\quad - \Delta \mathcal{F}_r + \frac{2}{r^2} \left[\mathcal{F}_r + \frac{1}{\sin\theta} \frac{\partial (\sin\theta \, \mathcal{F}_\theta)}{\partial \theta} + \frac{1}{\sin\theta} \frac{\partial \mathcal{F}_\phi}{\partial \phi} \right] , \\
&= \frac{1}{r^2} \frac{\partial}{\partial r} \left[\frac{r}{\sin\theta} \left(\frac{\partial (\sin\theta \, \mathcal{F}_\theta)}{\partial \theta} + \frac{\partial \mathcal{F}_\phi}{\partial \phi} \right) \right] - \Delta_H \, \mathcal{F}_r \,.
\end{aligned}$$

To make use of the recurrence relations (4.12)-(4.15), we then follow the same steps than for the nonlinear terms that enter the equation for poloidal potential of the magnetic field g:

$$\frac{1}{r^2} \frac{\partial}{\partial r} \left[r^2 \left(\vartheta_1 \, \mathcal{F}t_\ell^m + \frac{\partial \, \mathcal{F}p_\ell^m}{\partial \phi} \right) \right] + L_H \, \mathcal{F}r_\ell^m \,.$$

We thus finally get

$$\boxed{\mathcal{N}_{\ell m}^h = \ell(\ell+1)\,\mathcal{F}r_\ell^m + \frac{1}{r^2}\frac{\partial}{\partial r}\left[r^2\left\{(\ell+1)\,c_\ell^m\,\mathcal{F}t_{\ell-1}^m - \ell\,c_{\ell+1}^m\,\mathcal{F}t_{\ell+1}^m + im\,\mathcal{F}p_\ell^m\right\}\right]\,.} \tag{4.41}$$

See also:

The θ and ϕ derivatives that enter (4.41) are computed in the subroutine `get_td`. The remaining radial derivative is computed afterwards at the very beginning of `updateB`.

4.7 Boundary conditions and inner core

4.7.1 Mechanical boundary conditions

Since the system of equations is formulated on a radial grid, boundary conditions can simply be satisfied by replacing the collocation equation at grid points r_i and r_o with appropriate expressions. The condition of zero radial flow on the boundaries implies that the poloidal potential has to vanish, i.e. $W(r_o) = 0$ and $W(r_i) = 0$. In Chebychev representation this implies

$$\mathcal{C}_n(r)W_{\ell mn} = 0 \;\text{ at }\; r = r_i, r_o \;\;. \tag{4.42}$$

Note that the summation convention with respect to radial modes n is used again. **The no-slip** condition further requires that the horizontal flow components also have to vanish, provided the two boundaries are at rest. This condition is fulfilled for

$$\frac{\partial W}{\partial r} = 0 \;\text{ and }\; Z = 0,$$

at the respective boundary. In spectral space these conditions read

$$\mathcal{C}_n'(r)W_{\ell mn} = 0 \;\text{ at }\; r = r_i, r_o\,, \tag{4.43}$$

and

$$\mathcal{C}_n(r)Z_{\ell mn} = 0 \;\text{ at }\; r = r_i, r_o\,, \tag{4.44}$$

for all spherical harmonic modes (ℓ, m). The conditions (4.42)-(4.44) replace the poloidal flow potential equations (4.20) and the pressure equation (4.22), respectively, at the collocation points r_i and r_o.

If the inner-core and/or the mantle are allowed to react to torques, a condition based on the conservation of angular momentum replaces condition (4.44) for the mode $(\ell = 1, m = 0)$:

$$\mathsf{I}\frac{\partial \boldsymbol{\omega}}{\partial t} = \boldsymbol{\Gamma}\;\;.$$

The tensor I denotes the moment of inertia of inner core or mantle, respectively, $\boldsymbol{\omega}$ is the mantle or inner-core rotation rate relative to that of the reference frame, and $\boldsymbol{\Gamma}$ is the respective torque.

Free-slip boundary conditions require that the viscous stress vanishes, which in turn implies that the non-diagonal components $\mathsf{Sr}_{r\phi}$ and $\mathsf{S}_{r\theta}$ of the rate-of-strain tensor vanish. Translated to the spectral representation this requires

$$\left[\mathcal{C}_n''(r) - \left(\frac{2}{r} + \frac{d\ln\tilde{\rho}}{dr}\right)\mathcal{C}_n'(r)\right]W_{\ell mn} = 0 \;\text{ and }\; \left[\mathcal{C}_n'(r) - \left(\frac{2}{r} + \frac{d\ln\tilde{\rho}}{dr}\right)\mathcal{C}_n(r)\right]Z_{\ell mn} = 0\,.$$

We show the derivation for the somewhat simpler Boussinesq approximation which yields the condition

$$\frac{\partial}{\partial r}\frac{\boldsymbol{u}_H}{r} = 0$$

where the index H denotes the horizonal flow components. In terms of poloidal and toroidal components this implies

$$\frac{\partial}{\partial r}\frac{1}{r}\left(\boldsymbol{\nabla}_H \frac{\partial W}{\partial r}\right) = \boldsymbol{\nabla}_H \frac{1}{r}\left(\frac{\partial^2}{\partial r^2} - \frac{2}{r}\frac{\partial}{\partial r}\right) W = 0$$

and

$$\frac{\partial}{\partial r}\frac{1}{r}\nabla \times \boldsymbol{e}_r Z = \nabla \times \boldsymbol{e}_r \frac{1}{r}\left(\frac{\partial}{\partial r} - \frac{2}{r}\right) Z = 0$$

which can be fulfilled with

$$\left(\frac{\partial^2}{\partial r^2} - \frac{2}{r}\frac{\partial}{\partial r}\right) W = 0$$

and

$$\left(\frac{\partial}{\partial r} - \frac{2}{r}\right) Z = 0 \, .$$

In spectral representation this then reads

$$\left(\mathcal{C}_n'' - \frac{2}{r}\mathcal{C}_n'\right) W_{\ell mn} = 0 \;\text{ and }\; \left(\mathcal{C}_n' - \frac{2}{r}\mathcal{C}_n\right) Z_{\ell mn} = 0 \, .$$

4.7.2 Thermodynamic boundary conditions

For Entropy or temperature in the Boussinesq approximation either fixed value of fixed flux conditions are used. The former implies

$$s = \text{const.} \;\text{ or }\; T = \text{const.}$$

at r_i and/or r_o, while the latter means

$$\frac{\partial}{\partial r} s = \text{const.} \;\text{ or }\; \frac{\partial}{\partial r} T = \text{const.}$$

In spectral representation for example the respective entropy condition read

$$\mathcal{C}_n s_{\ell mn} = \text{const.} \;\text{ or }\; \mathcal{C}'_n s_{\ell mn} = \text{const.}$$

Appropriate constant values need to be chosen and are instrumental in driving the dynamo when flux conditions are imposed.

4.7.3 Magnetic boundary conditions and inner core

Three different magnetic boundary conditions are implemented in MagIC. The most simple one is the conceptual condition at the boundary to an infinite conductor. Surface current in this conductor will prevent the internally produced magnetic field from penetrating so that the field has to vanish at the boundary. The condition are thus the same as for a rigid flow (with boundaries at rest). We only provide the spectral representation here:

$$\mathcal{C}_n(r) W_{\ell mn} = 0 \;\text{ at }\; r = r_i, r_o \;\; . \tag{4.45}$$

Note that the summation convention with respect to radial modes n is used again. **The no-slip** condition further requires that the horizontal flow components also have to vanish, provided the two boundaries are at rest. This condition is fulfilled for

$$\mathcal{C}_n(r) g_{\ell mn} = 0 \;\;,\;\; \mathcal{C}'_n(r) g_{\ell mn} = 0 \;\text{ and }\; \mathcal{C}_n(r) h_{\ell mn} = 0. \tag{4.46}$$

More complex are the conditions to an electrical insulator. Here we actually use matching condition to a potential field condition that are formulated like boundary conditions. Since the electrical currents have to vanish in the insulator we have $\nabla \times \boldsymbol{B}$, which means that the magnetic field is a potential field $\boldsymbol{B}^I = -\boldsymbol{\nabla} V$ with $\Delta V = 0$. This Laplace equation implies a coupling between radial and horizontal derivatives which is best solved in spectral space. Two potential contributions have to be considered depending whether the field is produced above the interface radius r_{BC} or below. We distinguish these contributions with upper indices I for internal or below and E for external or above. The total potential then has the form:

$$V_{\ell m}(r) = r_{BC} V^I_{\ell m} \left(\frac{r_{BC}}{r}\right)^{\ell+1} + r_{BC} V^E_{\ell m} \left(\frac{r}{r_{BC}}\right)^{\ell}.$$

with the two spectral potential representations $V^I_{\ell m}$ and $V^E_{\ell m}$. This provides well defined radial derivative for both field contributions. For boundary r_o we have to use the first contribution and match the respective field as well as its radial derivative to the dynamo solution. The toroidal field cannot penetrate the insulator and thus simply vanishes which yields $h = 0$ or

$$\mathcal{C}_n h_{\ell mn} = 0$$

in spectral space. The poloidal field then has to match the potential field which implies

$$\nabla_H \frac{\partial}{\partial r} g = -\nabla_H V^I$$

for the horizontal components and

$$\frac{\nabla^2_H}{r^2} g = \frac{\partial}{\partial r} V^I$$

for the radial. In spectral space these condition can be reduce to

$$\mathcal{C'}_n(r) g_{\ell mn} = V^I_{lm} \quad \text{and} \quad \frac{\ell(\ell+1)}{r^2} \mathcal{C}_n g_{\ell mn} = -\frac{\ell+1}{r} V^I_{lm}.$$

Combining both allows to eliminate the potential and finally leads to the spectral condition used in MagIC:

$$\left(\mathcal{C'}_n(r_o) + \frac{\ell}{r_o}\mathcal{C}_n(r_o)\right) g_{\ell mn} = 0$$

Analogous consideration lead to the respective condition at the interface to an insulating inner core:

$$\left(\mathcal{C'}_n(r_i) - \frac{\ell+1}{r_i}\mathcal{C}_n(r_i)\right) g_{\ell mn} = 0.$$

If the inner core is modelled as an electrical conductor, a simplified dynamo equation has to be solved in which the fluid flow is replaced by the solid-body rotation of the inner core. The latter is described by a single toroidal flow mode ($\ell = 1, m = 0$). The resulting nonlinear terms can be expressed by a simple spherical harmonic expansion, where the superscript I denotes values in the inner core and ω_I its differential rotation rate:

$$\int d\Omega \, Y_\ell^{m\star} \, \boldsymbol{e_r} \cdot \left[\boldsymbol{\nabla} \times \left(\boldsymbol{u^I} \times \boldsymbol{B^I}\right)\right] = -i\,\omega_I\, m\, \frac{\ell(\ell+1)}{r^2}\, g^I_{\ell m}(r)\,, \tag{4.47}$$

$$\int d\Omega \, Y_\ell^{m\star} \, \boldsymbol{e_r} \cdot \left[\boldsymbol{\nabla} \times \boldsymbol{\nabla} \times \left(\boldsymbol{u^I} \times \boldsymbol{B^I}\right)\right] = -i\,\omega_I\, m\, \frac{\ell(\ell+1)}{r^2}\, h^I_{\ell m}(r)\,. \tag{4.48}$$

The expensive back and forth transformations between spherical harmonic and grid representations are therefore not required for advancing the inner-core magnetic field in time.

In the inner core the magnetic potentials are again conveniently expanded into Chebyshev polynomials. The Chebyshev variable x spans the whole diameter of the inner core, so that grid points are dense near the inner-core boundary but sparse in the center. The mapping is given by:

$$x(r) = \frac{r}{r_i} \quad , \quad -r_i \leq r \leq r_i \quad . \tag{4.49}$$

Each point in the inner core is thus represented twice, by grid points (r, θ, ϕ) and $(-r, \pi - \theta, \phi + \pi)$. Since both representations must be identical, this imposes a symmetry constraint that can be fulfilled when the radial expansion comprises only polynomials of even order:

$$g^I_{\ell m}(r) = \left(\frac{r}{r_i}\right)^{\ell+1} \sum_{i=0}^{M-1} g^I_{\ell m\, 2i}\, \mathcal{C}_{2i}(r) \quad . \tag{4.50}$$

An equivalent expression holds for the toroidal potential in the inner core. FFTs can again by employed efficiently for the radial transformation, using the M extrema of $\mathcal{C}_{2M-1}(r)$ with $x > 0$ as grid points.

The sets of spectral magnetic field equations for the inner and the outer core are coupled via continuity equations for the magnetic field and the horizontal electric field. Continuity of the magnetic field is assured by (**i**) continuity of the toroidal potential, (**ii**) continuity of the poloidal potential, and (**iii**) continuity of the radial derivative of the latter. Continuity of the horizontal electric field demands (**iv**) that the radial derivative of the toroidal potential is continuous, provided that the horizontal flow and the electrical conductivity are continuous at the interface. These four conditions replace the spectral equations (4.25), (4.26) on the outer-core side and equations (4.47), (4.48) on the inner-core side. Employing free-slip conditions or allowing for electrical conductivity differences between inner and outer core leads to more complicated and even non-linear matching conditions.

CHAPTER

FIVE

CONTRIBUTING TO THE CODE

MagIC is an open-source code, we thus value any possible contribution! There are several ways to directly contribute to the code:

Contribute

- **Do you want to contribute to the code?** Just clone the code and start modyfing it. Make sure that your modifications *don't alter the code*, try to *document your changes* as much as you can and follow the recommended *Fortran coding style*.
- **Do you want to improve the documentation?** Feel free to document some missing features. The documentation is stored in the directory `$MAGIC_HOME/doc/sphinx` and relies on the documenting tool Sphinx. Some recommendations regarding documentation can be found *below*.
- **Did you find a bug?** Issues and feature requests should be raised in the github tracker.

5.1 Checking the consistency of the code

It is frequently required to check the consistency of the code, especially **after the implementation of new features**. For this reason, we have the Perl script `magic_checks.pl`, located in the directory `$MAGIC_HOME/samples/`, which tests the compilation of the code and it's results against a set of standard solutions in sample directories to check if the code produces the correct output. It has been initially ported from the auto-test subroutines of the pencil-code developed by W. Dobler and adapted to the MagIC code.

You can run it as follows:

```
./magic_checks.pl <options>
```

It supports the following options:

```
-h,  --help              Show usage overview
-c,  --clean             Clean the directories when it is finished
-a,  --all               All auto-tests are computed
     --level=LEV         Run only tests from level LEV
     --max-level=LEV     Run all tests below with level <= LEV (default: 0)
     --hybrid            Run the hybrid version
     --use-cmake         Use CMake to build the code instead of make
     --use-mkl           Use the Math Kernel Library for FFTs and Lapack calls
```

Note: When using the `--use-cmake` option, make sure that your environment variables `FC` and `CC` are correctly defined

The `level=LEV` defines the priority level of check and validation of the code. It has the following levels of checking:

Level	Cases to check (subdirectories)
0	• Boussinesq dynamo benchmark (Christensen et al., 2001) - start from zero (`dynamo_benchmark`) • Variable transport properties (viscosity, thermal diffusivity and electrical diffusivity) in an anelastic convective model (`varProps`) • Boussinesq dynamo benchmark (Christensen et al., 2001) - start from a saturated state (`boussBenchSat`)
1	• Test reading and writing of restart files (`testRestart`) • Test different grid truncations (`testTruncations`) • Test mapping on to a new grid (`testMapping`) • Test different outputs produced (`testOutputs`) • Test different radial outputs - `*R.TAG` (`testRadialOutputs`)
2	• Hydrodynamic anelastic benchmark (Jones et al., 2011) (`hydro_bench_anel`)
3	• Heat flux perturbation (`fluxPerturbation`) • Isothermal model with $N_\rho = 3$ (`isothermal_nrho3`) • Boussinesq Dynamo benchmark for conducting and rotating inner core (`dynamo_benchmark_condICrotIC`) • Anelastic dynamo with variable conductivity (`varCond`)
4	• Test the writing of RMS force balance (`testRMSOutputs`) • Test the writing of Graphic and Movie files (`testGraphMovieOutputs`) • Test the writing of TO and Geos outputs (`testTOGeosOutputs`)

5.2 Advices when contributing to the code

- Before commiting your modifications **always** make sure that the auto-tests pass correctly.
- Try to follow the same coding style rules as in the rest of the code:
  1. **Never** use TABS but always SPACES instead
  2. Use 3 spaces for indentation

     Note: These two rules can be easily set in your $HOME/.vimrc file if you use vim:

```
au FileType fortran set shiftwidth=3
au FileType fortran set tabstop=3
au FileType fortran set expandtab
```

  3. Never use capital letters for variable declaration
  4. Never use `dimension(len)` for declaring array but rather `real(cp) :: data(len)`
  5. Always use the default precisions when introducing new variables `(cp)`

These rules try to follow the general recommendations on modern fortran programming that can be found on www.fortran90.org or in the book Modern Fortran - style and usage by N. S. Clerman and W. Spector.

5.3 Building the documentation and contributing to it

The documentation is generated using Sphinx. To build it you'll thus need to install this python module on your machine. This is in general directly available on most of the Linux distributions under the name `python-sphinx`. Once installed, just go to the documentation directory

```
$ cd $MAGIC_HOME/doc/sphinx
```

and build the html documentation

```
$ make html
```

The complete documentation will then be built in a local directory named `$MAGIC_HOME/doc/sphinx/.build/html`.

If LaTeX is installed on your work station, it is also possible to build the corresponding manual of the documentation in the pdf format:

```
$ make latexpdf
```

The resulting pdf is then generated in a local directory named `$MAGIC_HOME/doc/sphinx/.build/latex`.

It is pretty straightforward to contribute to the documentation by simply adding some contents to the different `rst` files. Informations about reStructuredText syntax can be found on www.sphinx-doc.org, while helpful CheatSheet are accessible here or there.

CHAPTER

SIX

INPUT PARAMETERS

True runtime input parameters are read from STDIN as namelists, a Fortran feature. A namelist is identified by its unique name *&name*. The name-statement is followed by the parameters that are part of the namelist in the format *parameter=value,*. The namelist is closed by a backslash. The subroutine *defaultNamelists* (in the module *Namelist.f90*) defines a default value for each parameter. Only the parameters whose value should differ from its default have to be stated in the namelist.

An example for the short namelist defining inner core parameters is

```
&inner_core
  sigma_ratio = 1.0,
  nRotIc      = 1
```

Comas can be used to seperate namelist entries since they are not interpreted by the code.

Magic uses the following **eight namelists** :

Namelists

1. *&grid* for resolution
2. *&control* for control parameters and numerical parameters.
3. *&phys_param* for the physical parameters.
4. *&B_external* for setting up an external field contribution
5. *&start_field* to define the starting fields.
6. *&output_control* for defining the output.
7. *&mantle* for setting mantle parameters.
8. *&inner_core* for setting inner core parameters.

The number of possible input parameters has grown to more than 100/150. **Don't be confused by all the possible options though, since all parameters are internally set to a useful default value!**

Practically, in a production run, the number of parameters you may want to adjust is thus much smaller. As an example, the following namelist shows you how to initiate and quickly run one of the anelastic benchmarks by (Jones et al., 2011):

```
&grid
 n_r_max     =97,           ! 97 radial grid points
 n_cheb_max  =95,
 n_phi_tot   =288,          ! 288 points in the azimuthal direction
 n_r_ic_max  =17,
n_cheb_ic_max=15,
 minc        =1,            ! No azimuthal symmetry
/
&control
 mode        =1,            ! This is a non-magnetic case
```

```
 tag         ="test",
 n_time_steps=50000,         ! Number of time steps
 courfac     =2.5D0,
 alffac      =1.0D0,
 dtmax       =1.0D-4,        ! Maximum allowed time-step
 n_cour_step =5,
 alpha       =0.6D0,
 runHours    =23,            ! Run time (hours)
 runMinutes  =30,            ! Run time (minutes)
/
&phys_param
 ra          =1.48638035D5,  ! Rayleigh number
 ek          =1.0D-3,        ! Ekman number
 pr          =1.0D0,         ! Prandtl number
 strat       =5.D0,          ! Density contrast
 polind      =2.0D0,         ! Polytropic index
 radratio    =0.35D0,        ! Aspect ratio of the spherical shell
 g0          =0.D0,          ! Gravity profile
 g1          =0.D0,
 g2          =1.D0,
 ktops       =1,             ! Entropy boundary condition
 kbots       =1,
 ktopv       =1,             ! Mechanical boundary condition
 kbotv       =1,
/
&start_field
 l_start_file=.FALSE.,
 start_file  ="rst_end.CJ3",
 init_s1     =1919,          ! Initial entropy perturbation pattern
 amp_s1      =0.01,          ! Amplitude of the initial perturbation
/
&output_control
 n_log_step  =50,            ! Store time series every 50 time steps
 n_graphs    =1,             ! 1 G_#.TAG file produced at the end of the run
 n_specs     =5,             ! 5 spectra produced during the run
 n_rsts      =1,             ! 1 rst_end.TAG file produced at the end of the run
 runid       ="C.Jones bench",
/
&mantle
 nRotMa      =0
/
&inner_core
 sigma_ratio =0.d0,          ! Non-conducting inner core
 nRotIC      =0,
/
```

This example might then be easily adapted to your desired configuration.

6.1 Grid namelist

This namelist defines the resolution of the computations. Keep in mind that **MagIC** is a 3D pseudo-spectral spherical shell code using Chebyshev polynomial expansions in the radial and spherical harmonic expansions in the angular directions.

6.1.1 Outer Core

- **n_r_max** (default *n_r_max=33*) is an integer which gives the number of grid points in the radial direction in the outer core ($[r_i, r_o]$). It must be of the form `4*n+1`, where `n` is an integer.

Note: The possible values for *n_r_max* are thus: 17, 21, 25, 33, 37, 41, 49, 61, 65, 73? 81, 97, 101, 121, 129, 145, 161, 257, 401, 513, ...

- **n_cheb_max** (default *n_cheb_max=31*) is an integer which is the number of terms in the Chebyshev polynomial expansion to be used in the radial direction - the highest degree of Chebyshev polynomial used being `n_cheb_max-1`. Note that `n_cheb_max <= n_r_max`.

Note: Adopting `n_cheb_max=n_r_max-2` is usually a good choice

- **n_phi_tot** (default *n_phi_tot=192*) is an integer which gives the number of longitudinal/azimuthal grid points. It has the following contraints:
  - `n_phi_tot` ' must be a multiple of `minc` (see below)
  - `n_phi_tot/minc` must be a multiple of 4
  - *n_phi_tot* must be a multiple of 16

Note: The possible values for *n_phi_max* are thus: 16, 32, 48, 64, 96, 128, 192, 256, 288? 320, 384, 400, 512, 576, 640, 768, 864, 1024, 1280, 1536, 1792, 2048, ...

6.1.2 Inner Core

- **n_r_ic_max** (default *n_r_ic_max=17*) is an integer which gives the number of grid points in the radial direction in the inner core ($[0, r_i]$). It too, must be of the form `4*n+1`, where `n` is an integer.
- **n_cheb_ic_max** (default *n_cheb_ic_max=15*) is the number of terms in the Chebyshev polynomial expansion in the radial direction in the inner core. Only Chebyshev polynomials of even degrees are used in the expansion giving the highest degree used to be `2*n_cheb_ic_max-2`. Note that here too, `n_cheb_ic_max <= n_r_max`.

6.1.3 Symmetry and aliasing

- **minc** (default *minc=1*) is an integer which gives the longitudinal symmetry. e.g: `minc=n` would give an n-fold rotational symmetry in the azimuthal direction. One can use this to reduce computational costs when the symmetry of the solution is known. The orders of the spherical harmonic expansion (`m`) are multiples of *minc*.
- **nalias** (default *nalias=20*) is an integer which determines antialiasing used in the spherical harmonic representation. Note that `20 <= nalias <= 30`.

The number of grid points in latitude *n_theta_max = n_phi_tot/2*. The maximum degree (*l_max*) and maximum order (*m_max*) of the spherical harmonic expansion are determined by *nalias*:

```
l_max = (nalias * n_theta_max)/30
```

6.2 Control namelist

This namelist defines the numerical parameters of the problem plus the variables that control and organize the run.

- **mode** (default `mode=0`) is an integer which controls the type of calculation performed.

mode=0	Self-consistent dynamo
mode=1	Convection
mode=2	Kinematic dynamo
mode=3	Magnetic decay modes
mode=4	Magneto convection
mode=5	Linear onset of convection
mode=6	Self-consistent dynamo, but with no Lorentz force
mode=7	Super-rotating inner core or mantle, no convection and no magnetic field
mode=8	Super-rotating inner core or mantle, no convection
mode=9	Super-rotating inner core or mantle, no convection and no Lorentz force
mode=10	Super-rotating inner core or mantle, no convection, no magnetic field, no Lorentz force and no advection

- **tag** (default `tag="default"`) is a character string, used as an extension for all output files.
- **n_time_steps** (default `n_time_steps=100`) is an integer, the number of time steps to be performed.
- **tEND** (default `tEND=0.0`) is a real, which can be used to force the code to stop when :math:`t=tEND`. This is only used when `t/=tEND`.
- **alpha** (default `alpha=0.5`) is a real. This is the weight used for current time step in implicit time step.

6.2.1 Default scales

- **n_tScale** (default `n_tScale=0`) is an integer, which determines the time scaling

n_tScale=0	Use viscous time scale.
n_tScale=1	Use magnetic time scale.
n_tScale=2	Use thermal time scale.

- **n_lScale** (default `n_lScale=0`) is an integer which determines the reference length scale.

n_lScale=0	Use outer core.
n_lScale=1	Use total core.

- **enscale** (default `enscale=1.0`) is a real. This is the scaling for energies.

6.2.2 Update control

- **l_update_v** (default `l_update_v=.true.`) is a logical that specifies whether the velocity field should be time-stepped or not.
- **l_update_b** (default `l_update_b=.true.`) is a logical that specifies whether the magnetic field should be time-stepped or not.
- **l_update_s** (default `l_update_s=.true.`) is a logical that specifies whether the entropy/temperature should be time-stepped or not.

6.2.3 Time step control

A modified courant criteria including a modified Alfven-velocity is used to account for the magnetic field. The relative and absolute importance of flow and Alfven-velocity can be controled by **courfac** and **alffac** respectively.

- **dtstart** (default `dtstart=0.0`) is a real, which is used as the initial time step if the starting solution is initialized (see below) and dtstart > 0.

- **dtMax** (default *dtMax=1e-4*) is a real. This is the maximum allowed time step δt. If $\delta t >$ dtmax, the time step is decreased to at least dtmax (See routine *dt_courant*). Run is stopped if $\delta t <$ dtmin and dtmin $= 10^{-6}$ dtmax.
- **courfac** (default *courfac=2.5*) is a real used to scale velocity in courant criteria.
- **alffac** (default *alffac=1.0*) is a real, used to scale Alfven-velocity in courant criteria.
- **n_cour_step** (default *n_cour_step=10*) is an integer. This is the number of time steps before consecutive checking of courant criteria. Note: the courant criteria is checked always after the time step has been changed if `n_cour_step>0`.

6.2.4 Run time

The total desired runtime (in human units and not in CPU units) can be specified with the three variables **runHours**, **runMinutes** and **runSeconds**.

- **runHours** (default `runHours=0`) is an integer that controls the number of run hours.
- **runMinutes** (default `runMinutes=0`) is an integer that controls the .
- **runSeconds** (default `runSeconds=0`) is an integer that controls the number of run hours.

Here is an example for a run of 23h30:

```
runHours   = 23,
runMinutes = 30,
```

6.2.5 Hyperdiffusivity

Hyperdiffusion can be applied by multiplying the diffusion operators by a factor of the form

$$d(\ell) = 1 + D \left[\frac{\ell + 1 - \ell_{hd}}{\ell_{max} + 1 - \ell_{hd}} \right]^{\beta}$$

for the spherical harmonic degrees $\ell \geq \ell_{hd}$.

- **difnu** (default *difnu=0.0*) is a real. This is the amplitude D of the viscous hyperdiffusion.
- **difkappa** (default `difkappa=0.0`) is a real. This is the amplitude D of the thermal hyperdiffusion.
- **difeta** (default *difeta=0.0*) is a real. This is the amplitude D of the magnetic hyperdiffusion.
- **ldif** (default *ldif=1*) is an integer. This is the degree ℓ_{hd} where hyperdiffusion starts to act.
- **ldifexp** (default *ldifexp=-1*) is an integer. This is the exponent β of hyperdiffusion.

6.2.6 Angular momentum correction

In case of the use of stress-free boundary conditions at both boundaries, it is safer to ensure that the angular momentum is correctly conserved. This can be enforced through the following input variables:

- **l_correct_AMe** (default *l_correct_AMe=.false.*) is a logical. This is used to correct the equatorial angular momentum.
- **l_correct_AMz** (default *l_correct_AMz=.false.*) is a logical. This is used to correct the axial angular momentum.

6.2.7 Mapping of the Gauss-Lobatto grid

Non-linear mapping function to concentrate/diperse grid points around a point inside the domain. For full description, see Bayliss and Turkel (1990). The function that re-distributes the collocation points is

$$r = \alpha_2 + \left[\frac{\tan^{-1}\left(\lambda(r_{cheb} - x_0)\right)}{\alpha_1} \right] ,$$

where the Gauss-Lobatto collocation points are

$$r_{cheb} = \cos\left(\frac{\pi(k-1)}{n_r}\right) , \quad k = 1, 2, ..., n_r , \; n_r = n_r_max$$

and $r \in [r_i, r_o]$, $r_{cheb} \in [-1.0, 1.0]$. The parameters to calculate r are

$$\begin{aligned} \lambda &= \frac{\tan^{-1}\left(\alpha_1(1-\alpha_2)\right)}{1 - x_0} \\ x_0 &= \frac{K-1}{K+1} \\ K &= \frac{\tan^{-1}\left(\alpha_1(1+\alpha_2)\right)}{\tan^{-1}\left(\alpha_1(1-\alpha_2)\right)} . \end{aligned}$$

The coefficient α_1 determines the degree of concentration/dispersion of the grid points around $r_{cheb} = \alpha_2$. If α_1 is too high, the r function becomes nearly discontinuous. To avoid numerical problems, α_1 should remain close to unity.

- **l_newmap** (default `l_newmap=.false.`) is a logical. A radial mapping can be applied to the Chebyshev grid when `l_newmap` is set to `.true.`. The radial profile of the mapping function is then stored during the initialisation of the code in the file *rNM.TAG*.
- **alph1** (default `alph1=2.0`) is a real. This is a control parameter of the mapping function.
- **alph2** (default `alph2=0.0`) is a real. This is a control parameter of the mapping function.

6.2.8 Miscellaneous

- **l_non_rot** (default `l_non_rot=.false.`) is a logical. Use it when you want to do non-rotating numerical simulations.

6.3 Physical parameters namelist

This namelist contains all the appropriate relevant control physical parameters.

6.3.1 Dimensionless control parameters

- **ra** (default `ra=1.1e5`) is a real. This the Rayleigh number expressed by

$$Ra = \frac{\alpha g_o \Delta T d^3}{\kappa \nu}$$

- **ek** (default `ek=1e-3`) is a real. This is the Ekman number expressed by

$$E = \frac{\nu}{\Omega d^2}$$

- **pr** (default `pr=1.0`) is a real. This is the Prandtl number expressed by

$$Pr = \frac{\nu}{\kappa}$$

- **prmag** (default `prmag=5.0`) is a real. This is the magnetic Prandtl number expressed by

$$Pm = \frac{\nu}{\lambda}$$

- **radratio** (default `radratio=0.35`) is a real. This is the ratio of the inner core radius r_i to the outer core radius r_o:

$$\eta = \frac{r_i}{r_o}$$

- **strat** (default `strat=0.0`) is a real. This is the number of density scale heights of the reference state:

$$N_\rho = \ln \frac{\tilde{\rho}(r_i)}{\tilde{\rho}(r_o)}$$

- **polind** (default `polind=1.5`) is a real. This is the polytropic index, which relates the background temperature to the background density:

$$\tilde{\rho} = \tilde{T}^m$$

> **Warning:** Be careful: in its current version the code only handles **adiabatic** backgrounds, therefore changing `polind` physically means that the nature of the fluid (in particular its Grüneisen parameter) will change. For an ideal gas, it actually always follows $m + 1 = \frac{\gamma - 1}{\gamma}$

- **l_isothermal** (default `l_isothermal=.false.`) is a logical. When set to `.true.`, makes the temperature background isothermal (i.e. $\tilde{T} = cst.$). In that case, the dissipation number Di vanishes and there is no viscous and Ohmic heating left. The only difference with the Boussinesq set of equations are thus restricted to the density background $\tilde{\rho}$ and its radial derivatives that enters the viscous stress. This approximation is also called the **zero Grüneisen parameter** and was extensively explored by Denise Tortorella during her PhD.

6.3.2 Heat sources and sinks

- **epsc0** (default `epsc0=0.0`) is a real. This is the volumetric heat source ϵ_0 that enters the thermal equilibrium relation:

$$-\nabla \cdot \left(\tilde{\rho} \tilde{T} \nabla s \right) + \epsilon_0 \, f(r) = 0 \tag{6.1}$$

  The radial function $f(r)$ can be modified with the variable `nVarEps` that enters the same input namelist.

- **nVarEps** (default `nVarEps=0`) is an integer. This is used to modify the radial-dependence ofthe volumetric heat source, i.e. $f(r)$ that enters equation (6.1).

`nVarEps=0`	Constant, i.e. $f(r) =$ cst..
`nVarEps=1`	Proportional to density, i.e. $f(r) = \tilde{\rho}(r)$.

6.3.3 Realistic interior models

- **interior_model** (default *interior_model="None"*) is a character string. This defines a polynomial fit of the density profile of the interior structure of several astrophysical objects. Possible options are "earth", "jupiter", "saturn" and "sun" (the naming is **not** case sensitive).

  Warning: When interior_model is defined the variables strat, polind, g0, g1 and g2 are not interpreted.

  The subroutine *radial* gives the exact details of the implementation.

- **r_cut_model** (default *r_cut_model=0.98*) is a real. This defines the cut-off radius of the reference model, i.e. the fluid domain is restricted to radii with $r \leq r_{cut}$.

The following input parameters will thus define a polynomial fit to the expected interior structure of Jupiter until 99% of Jupiter's radius (assumed here at the 1 bar level)

```
interior_model="JUP",
r_cut_model    =0.99e0,
```

6.3.4 Gravity

The radial dependence of the gravity profile can be adjusted following

$$g(r) = g_0 + g_1 \frac{r}{r_o} + g_2 \left(\frac{r_o}{r}\right)^2 \tag{6.2}$$

The three following parameters are used to set this profile

- **g0** (default *g0=0*) is the pre-factor of the constant part of the gravity profile, i.e. g_0 in equation (6.2).
- **g1** (default *g1=1*) is the pre-factor of the linear part of the gravity profile, i.e. g_1 in equation (6.2).
- **g2** (default *g2=0*) is the pre-factor of the $1/r^2$ part of the gravity profile, i.e. g_2 in equation (6.2).

6.3.5 Transport properties

- **difExp** (default *difExp=-0.5*) is a real. This is the exponent that is used when *nVarVisc=2*, *nVarDiff=2* or *nVarCond=4*.

Electrical conductivity

There are several electrical conductivity profiles implemented in the code that can be chosen with the *nVarCond* input variable. The following one corresponds to a constant electrical conductivity in the deep interior ($r < r_m$) and an exponential decay in the outer layer.

$$\begin{aligned} \sigma(r) &= 1 + (\sigma_m - 1)\left(\frac{r - r_i}{r_m - r_i}\right)^a \quad \text{for} \quad r < r_m, \\ \sigma(r) &= \sigma_m \exp\left[a\left(\frac{r - r_m}{r_m - r_i}\right)\frac{\sigma_m - 1}{\sigma_m}\right] \quad \text{for} \quad r \geq r_m. \end{aligned} \tag{6.3}$$

- **nVarCond** (default *nVarCond=0*) is an integer. This is used to modify the radial-dependence of the electrical conductivity.

`nVarCond=0`	Constant electrical conductivity, i.e. $\sigma = \text{cst}$.
`nVarCond=1`	$\sigma \propto \tanh[a(r - r_m)]$
`nVarCond=2`	See equation (6.3).
`nVarCond=3`	**Magnetic diffusivity proportional to $1/\tilde{\rho}$, i.e.** $$\lambda = \frac{\tilde{\rho}_i}{\tilde{\rho}}$$
`nVarCond=2`	**Radial profile of the form:** $$\lambda = \left(\frac{\tilde{\rho}(r)}{\tilde{\rho}_i}\right)^{\alpha}$$

- **con_RadRatio** (default *con_RadRatio=0.75*) is a real. This defines the transition radius r_m that enters equation (6.3).
- **con_DecRate** (default *con_DecRate=9*) is an integer. This defines the decay rate a that enters equation (6.3).
- **con_LambdaMatch** (default *con_LambdaMatch=0.6*) is a real. This is the value of the conductivity at the transition point σ_m that enters equation (6.3).
- **con_LambdaOut** (default *con_LambdaOut=0.1*) is a real. This is the value of the conduvity at the outer boundary. This parameter is only used when `nVarCond=1`.
- **con_FuncWidth** (default *con_FuncWidth=0.25*) is a real. This parameter is only used when `nVarCond=1`.
- **r_LCR** (default *r_LCR=2.0*) is a real. `r_LCR` possibly defines a low-conductivity region for $r \geq r_{LCR}$, in which the electrical conductivity vanishes, i.e. $\lambda = 0$.

Thermal diffusivity

- **nVarDiff** (default *nVarDiff=0*) is an integer. This is used to change the radial-dependence of the thermal diffusivity:

`nVarDiff=0`	Constant thermal diffusivity κ
`nVarDiff=1`	**Constant thermal conductivity, i.e.** $$\kappa = \frac{\tilde{\rho}_i}{\tilde{\rho}(r)}$$
`nVarDiff=2`	**Radial profile of the form:** $$\kappa = \left(\frac{\tilde{\rho}(r)}{\tilde{\rho}_i}\right)^{\alpha}$$
`nVarDiff=3`	polynomial-fit to an interior model of Jupiter
`nVarDiff=4`	polynomial-fit to an interior model of the Earth liquid core

Viscosity

- **nVarVisc** (default *nVarVisc=0*) is an integer. This is used to change the radial-dependence of the viscosity:

`nVarVisc=0`	Constant kinematic viscosity ν
`nVarVisc=1`	**Constant dynamic viscosity, i.e.** $$\nu = \frac{\tilde{\rho}_o}{\tilde{\rho}(r)}$$
`nVarVisc=2`	**Radial profile of the form:** $$\nu = \left(\frac{\tilde{\rho}(r)}{\tilde{\rho}_i}\right)^{\alpha}$$

where α is an exponent set by the namelist input variable `difExp`.

6.3.6 Anelastic liquid equations

Warning: This part is still work in progress. The input parameters here are likely to be changed in the future.

- **epsS** (default *epsS=0.0*) is a real. It controls the deviation to the adiabat. It can be related to the small parameter ϵ:

$$\epsilon \simeq \frac{\Delta T}{T} \simeq \frac{\Delta s}{c_p}$$

- **cmbHflux** (default *cmbHflux=0.0*) is a real. This is the CMB heat flux that enters the calculation of the reference state of the liquid core of the Earth, when the anelastic liquid approximation is employed.
- **slopeStrat** (default *slopeStrat=20.0*) is a real. This parameter controls the transition between the convective layer and the stably-stratified layer below the CMB.

6.3.7 Boundary conditions

Thermal boundary conditions

- **ktops** (default *ktops=1*) is an integer to specify the outer boundary entropy (or temperature) boundary condition:

`ktops=1`	Fixed entropy at outer boundary: $s(r_o) = s_{top}$
`ktops=2`	Fixed entropy flux at outer boundary: $\partial s(r_o)/\partial r = s_{top}$

- **kbots** (default *ktops=1*) is an integer to specify the inner boundary entropy (or temperature) boundary condition.
- **s_top** (default *s_top= 0 0 0.0 0.0*) is a real array of lateraly varying outer heat boundary conditions. Each four consecutive numbers are interpreted as follows:

  1. Spherical harmonic degree ℓ
  2. Spherical harmonic order m
  3. Real amplitude ($\cos$ contribution)
  4. Imaginary amplitude ($\sin$ contribution)

  For example, if the boundary condition should be a combination of an $(\ell = 1, m = 0)$ sherical harmonic with the amplitude 1 and an $(\ell = 2, m = 1)$ spherical harmonic with the amplitude (0.5,0.5) the respective namelist entry could read:

```
s_top = 1, 0, 1.0, 0.0, 2, 1, 0.5, 0.5, ! The comas could be left away.
```

- **s_bot** (default *s_bot=0 0 0.0 0.0*) is a real array. This is the same as s_top but for the bottom boundary.
- **impS** (default *impS=0*) is an integer. This is a flag to indicate if there is a localized entropy disturbance, imposed at the CMB. The number of these input boundary conditions is stored in n_impS (the maximum allowed is 20), and it's given by the number of sCMB defined in the same namelist. The default value of impS is zero (no entropy disturbance). If it is set in the namelist for an integer greater than zero, then sCMB has to be also defined in the namelist, as shown below.
- **sCMB** (default sCMB=0.0 0.0 0.0 0.0) is a real array of CMB heat boundary conditions (similar to the case of s_bot and s_top). Each four consecutive numbers are interpreted as follows:
  1. Highest amplitude value of the entropy boundary condition, stored in array *peakS(20)*. When impS<0, peakS is a relative amplitude in comparison to the $(\ell = 0, m = 0)$ contribution (for example, the case s_top= 0 0 -1 0).
  2. θ coordinate (input has to be given in degrees), stored in array *thetaS(20)*.
  3. ϕ coordinate (input has to be given in degrees), stored in array *phiS(20)*.
  4. Angular width (input has to be given in degrees), stored in array *widthS(20)*.

Mechanical boundary conditions

- **ktopv** (default *ktopv=2*) is an integer, which corresponds to the mechanical boundary condition for $r = r_o$.

ktopv=1	**Stress-free outer boundary for $r = r_o$:** $$W_{\ell m}(r = r_o) = 0, \quad \frac{\partial}{\partial r}\left(\frac{1}{r^2\tilde{\rho}}\frac{\partial W_{\ell m}}{\partial r}\right) = 0$$ $$\frac{\partial}{\partial r}\left(\frac{1}{r^2\tilde{\rho}} Z_{\ell m}\right) = 0$$
ktopv=2	**Rigid outer boundary for $r = r_o$:** $$W_{\ell m} = 0, \quad \frac{\partial W_{\ell m}}{\partial r} = 0,$$ $$Z_{\ell m} = 0$$

- **kbotv** (default *kbotv=2*) is an integer, which corresponds to the mechanical boundary condition for $r = r_i$.

Magnetic boundary conditions

- **ktopb** (default *ktopb=1*) is an integer, which corresponds to the magnetic boundary condition for $r = r_o$.

ktopb=1	**Insulating outer boundary:** $$\frac{\partial g_{\ell m}}{\partial r} + \frac{\ell}{r} g_{\ell m} = 0, \quad \frac{\partial h_{\ell m}}{\partial r} = 0$$
ktopb=3	Finitely conducting mantle
ktopb=4	**Pseudo-vacuum outer boundary:** $$\frac{\partial g_{\ell m}}{\partial r} = 0, \quad h_{\ell m} = 0$$

- **kbotb** (default *kbotb=1*) is an integer, which corresponds to the magnetic boundary condition for $r = r_i$.

kbotb=1	**Insulating inner boundary:** $$\frac{\partial g_{\ell m}}{\partial r} - \frac{\ell+1}{r} g_{\ell m} = 0, \quad \frac{\partial h_{\ell m}}{\partial r} = 0$$
ktopb=2	**Perfectly-conducting innner core:** $$g_{\ell m} = \frac{\partial g_{\ell m}}{\partial r} = 0, \quad \frac{\partial h_{\ell m}}{\partial r} = 0$$
ktopb=3	Finitely conducting innner core
ktopb=4	**Pseudo-vacuum outer boundary:** $$\frac{\partial g_{\ell m}}{\partial r} = 0, \quad h_{\ell m} = 0$$

6.4 External Magnetic Field Namelist

The namelist `&B_external` provides options for imposing an external magnetic field.

6.4.1 Externally imposed magnetic field

- **n_imp** (default *n_imp = 0*) is an integer controling the type of external field applied.

n_imp=0	No external magnetic field
n_imp=1	Follows idea of Uli Christensen of external field compensating internal field such that radial component of magnetic field vanishes at $r/r_{cmb} = rrMP$ where `rrMP` is the 'magnetopause radius' input by the user (see below)
n_imp=2	Uniform axisymmetric magnetic field of geometry given by `l_imp` (see below)
n_imp=3	Uniform axisymmetric magnetic field which changes direction according to the direction of the axial dipole of the internal magnetic field
n_imp=4	Same as `n_imp=3` but the amplitude of the external field is scaled to the amplitude of the axial dipole of the internal field
n_imp=7	External field depends on internal axial dipole through Special Heyner feedback functions

- **rrMP** (default *rrMP = 0.0*) is a real which gives the value of 'magnetopause radius'. In other words, it gives the radius (as a fraction of `r_cmb`) at which the radial component of the magnetic field vanishes due to cancelling out of external and internal magnetic field components. Used only when `n_imp = 1`.
- **amp_imp** (default *amp_imp = 0.0*) is a real which gives the amplitude of the external magnetic field.
- **expo_imp** (default *expo_imp = 0.0*) is a real which gives the exponent of dependence of external magnetic field on the axial dipole of the internal magnetic field. Used for `n_imp=7`.
- **bmax_imp** (default *bmax_imp = 0.0*) is a real which gives the location of the maximum of the ratio of the poloidal potentials g_{ext}/g_{int}.
- **l_imp** (default *l_imp = 1*) is an integer which gives the geometry (degree of spherical harmonic) of the external magnetic field. The external field is always axisymmetric, hence `m = 0` always. This option is used when `n_imp = 2,3` or `4`.

6.4.2 Current carrying loop

To simulate experiments, an external current carrying loop, concentric to the sphere and in the equatorial plane, has been implemented in the code. It's radius is fixed at a distance $a = r_{cmb}/0.8$ to match conditions of the Maryland 3 metre experiment.

- **l_curr** (default `l_curr = .false.`) is a logical that controls switching on or off of the current carrying loop.
- **amp_curr** (default `amp_curr = 0.0`) is a real that gives the amplitude of magnetic field produced by the current carring loop.

Warning: Note that an external magnetic field is incompatible with a region of low conductivity inside the spherical shell (i.e, if `r_LCR < r_cmb`). Thus, while imposing an external magnetic field, make sure `r_LCR > r_cmb` (which is the default case). For details on `r_LCR`, have a look at the section on *electrical conductivity* in the namelist for *physical parameters*.

6.5 Start field namelist

This namelist controls whether a start field from a previous solution should be used, or a specific field should be initialized.

6.5.1 Reading an input file of start fields

- **l_start_file** (default `l_start_file=.false.`) is a logical that controls whether the code should to read a file named `start_file` or not.
- **start_file** (default `start_file="no_start_file"`) is a character string. This is the name of the *restart file*.
- **inform** (default `inform=-1`) is an integer that can be used to specify the format of `start_file`. This ensures possible backward compatibility with previous versions of the code. You shouldn't change this value except to read very old *rst_end.TAG* files generated by older versions of MagIC.

`inform=0`	Oldest format used by U. Christensen
`inform=1`	Newer format used by U. Christensen
`inform=2`	Inner core introduced by J. Wicht
`inform=-1`	Default format

- **scale_s** (default `scale_s=1.0`) is a real. It can be possibly used to multiply the input entropy field from `start_file` by a constant factor `scale_s`.
- **scale_v** (default `scale_v=1.0`) is a real. It can be possibly used to multiply the input velocity field from `start_file` by a constant factor `scale_v`.
- **scale_b** (default `scale_b=1.0`) is a real. It can be possibly used to multiply the input magnetic field from `start_file` by a constant factor `scale_b`.
- **tipdipole** (default `tipdipole=0.0`) is a real that can be used to add non-axisymmetric disturbances to a start solution if non-axisymmetric parts have been lost due to mapping to a different symmetry. A $(\ell = 1, m = 1)$ entropy term is added with:

$$s_{10}(r) = \text{tipdipole}\ \sin[\pi(r - r_i)]$$

If a magnetic field without an $m = 1$ term is mapped into a field that permits this term, the code adds the respective poloidal field using the $(\ell = 1, m = 0)$ poloidal magnetic field and scaling it with `tipdipole`.

- **l_reset_t** (default *l_reset_t=.false.*) is a logical that can be set to `.true.` in case one wants to reset the time of start file to zero.

6.5.2 Defining the starting conditions

Initialisation of entropy

The heat equation with possible heat sources and sinks given by *epsc0* is solved for the spherically-symmetric term ($\ell = 0, m = 0$) to get its radial dependence. In addition to this initial state, two other laterally varying terms can be initialized. Their radial dependence are assumed to follow:

$$s(r) = 1 - 2\,x^2 + 3\,x^4 - x^6,$$

where

$$x = 2\,r - r_o - r_i\,.$$

The initial perturbation is thus set to zero at both boundaries r_i and r_o, and reaches its maximum amplitude of `amp_s1` or `amp_s2` at the mid-shell radius $r_i + 1/2$.

- **init_s1** (default *init_s1=0*) is an integer that controls the initial entropy. The following values are possible:
  - `init_s1=0`: nothing is initialized
  - `init_s1<100`: a random-noise of amplitude `amp_s1` is initialised. The subroutine `initS` in `init_fields.f90` gives the detail of this implementation.
  - `init_s1>100`: initialisation of mode with the spherical harmonic order m given by the last two (or three) digits of `init_s1` and the spherical harmonic degree ℓ given by the first two (or three) digits. Here are two examples:

```
init_s1   = 0707,
amp_s1    = 0.05,
```

    will introduce a perturbation on the mode ($\ell = 7, m = 7$) with an amplitude of 0.05.

```
init_s1   = 121121,
amp_s1    = 0.01,
```

    will introduce a perturbation on the mode ($\ell = 121, m = 121$) with an amplitude of 0.01.

- **amp_s1** (default *amp_s1=0.0*) is a real used to contol the amplitude of the perturbation defined by `init_s1`.
- **init_s2** (default *init_s2=0*) is an integer that controls a second spherical harmonic degee. It follows the same specifications as `init_s1`.
- **amp_s2** (default *amp_s2=0.0*) is a real used to contol the amplitude of the perturbation defined by `init_s2`.

Initialisation of magnetic field

- **init_b1** (default *init_b1=0*) is an integer that controls the initial magnetic field. The following values are possible:
  - `init_b1<0`: random noise initialization of all (ℓ, m) modes, except for ($\ell = 0, m = 0$). The subroutine `initB` in the file `init_fields.f90` contains the details of the implementation.
  - `init_b1=0`: nothing is initialized
  - `init_b1=1`: diffusive toroidal field initialized. Mode determined by `imagcon`.

  - `init_b1=2`: $(\ell = 1, m = 0)$ toroidal field with a maximum field strength of `amp_b1`. The radial dependence is defined, such that the field vanishes at both the inner and outer boundaries. In case of an insulating inner core: $h(r) \approx r \sin[\phi(r - r_o)]$. In case of a conducting inner core: $h(r) \approx r \sin[\pi(r/r_o)]$.
  - `init_b1=3`: $(\ell = 1, m = 0)$ poloidal field whose field strength is `amp_b1` at $r = r_i$. The radial dependence is chosen such that the current density j is independent of r:, i.e. $\partial j/\partial r = 0$. $(\ell = 2, m = 0)$ toroidal field with maximum strength `amp_b1`.
  - `init_b1=4`: $(\ell = 1, m = 0)$ poloidal field as if the core were an insulator (potential field). Field strength at $r = r_i$ is again given by `amp_b1`.
  - `init_b1=5`: $(\ell = 1, m = 0)$ poloidal field with field strength `amp_b1` at $r = r_i$. The radial dependence is again defined by $\partial j/\partial r = 0$.
  - `init_b1=6`: $(\ell = 1, m = 0)$ poloidal field independend of r.
  - `init_b1=7`: $(\ell = 1, m = 0)$ poloidal field which fulfills symmetry condition in inner core: $g(r) \approx \left(\frac{r}{r_i}\right)^2 \left[1 - \frac{3}{5}\left(\frac{r}{r_o}\right)^2\right]$. The field strength is given by `amp_b1` at $r = r_o$.
  - `init_b1=8`: same poloidal field as for `init_b1=7`. The toroidal field fulfills symmetry conditions in inner core and has a field strength of `amp_b1` at $r = r_i$: $h(r) \approx \left(\frac{r}{r_i}\right)^3 \left[1 - \left(\frac{r}{r_o}\right)^2\right]$.
  - `init_b1=9`: $(\ell = 2, m = 0)$ poloidal field, which is a potential field at the outer boundary.
  - `init_b1=10`: equatorial dipole only.
  - `init_b1=11`: axial and equatorial dipoles.
  - `init_b1=21`: toroidal field created by inner core rotation, equatorially symmetric $(\ell = 1, m = 0)$: $h(r) =$ ampb1 $\left(\frac{r_i}{r}\right)^6$. The field strength is given by `amp_b1` at $r = r_i$.
  - `init_b1=22`: toroidal field created by inner core rotation, equatorially antisymmetric $(\ell = 2, m = 0)$. Same radial function as for `init_b1=21`.
- **amp_b1** (default *amp_b1=0.0*) is a real used to contol the amplitude of the function defined by `init_b1`.
- **imagcon** (default *imagcon=0*) is an integer, which determines the imposed magnetic field for magnetoconvection. The magnetic field is imposed at boundaries.
  - `imagcon=0`: no magneto-convection
  - `imagcon<0`: axial poloidal dipole imposed at ICB with a maximum magnetic field strength `amp_b1`.
  - `imagcon=10`: $(\ell = 2, m = 0)$ toroidal field imposed at ICB and CMB with a maximum amplitude `amp_b1` at both boundaries.
  - `imagcon=11`: same as `imagcon=10` but the maximum amplitude is now `amp_b1` at the ICB and `-amp_b1` at the CMB.
  - `imagcon=12`: $(\ell = 1, m = 0)$ toroidal field with a maximum amplitude of `amp_b1` at the ICB and the CMB.
- **tmagcon** (*tmagcon=0.0*) is a real.

Initialisation of velocity field

- **init_v1** (default *init_v1=0*) is an integer that controls the initial velocity. The following values are possible:
  - `init_v1=0`: nothing is initialized

- `init_v1=1`: a differential rotation profile of the form

$$\Omega = \Omega_{ma} + 0.5\Omega_{ic} \quad \text{for} \quad s \leq r_i$$
$$\Omega = \Omega_{ma} \quad \text{for} \quad s > r_i$$

  where $s = r\sin\theta$ is the cylindrical radius. This profile only makes sense when one studies spherical Couette flows.

- `init_v1=2`: a differential rotation profile of the form $\Omega = \frac{\text{ampv1}}{\sqrt{1+s^4}}$ is introduced.
- `init_v1>2`: a random-noise of amplitude `amp_v1` is initialised. The subroutine `initV` in `init_fields.f90` gives the detail of this implementation.

- **amp_v1** (default *`amp_v1=0.0`*) is a real used to contol the amplitude of the function defined by `init_v1`.

6.6 Output control namelist

This namelist contains all the parameters that can be adjusted to control the outputs and diagnostics calculated by the code.

There are four different ways to control at which time step a specific output should be written. Outputs are generally distributed over the total calculation intervall unless an output time interval is defined by a start time `t_start` and a stop time `t_stop`. If no `t_start` is provided, the start time of the calculation is used. If no `t_stop` is provided or `t_stop>t_start` the total calculation interval is assumed

1. **Prescribed number of outputs**. The outputs are distributed evenly over the total calculation interval so that the number of timesteps between two outputs is always the same, with the possible exception of the first interval. Last output is written for the last time step, and to compensate the interval before the first output may be longer. However, if `t_stop` is provided, the outputs are distributed evenly over the interval [`t_stop`, `t_start`] with equal times intervals between them.

Note: These input variables are usually named with a pattern that follows `n_outputName`, for instance, *n_graphs*, *n_rsts*, *n_specs*, *n_logs*, etc.

In case you want to make use ot a specific time interval, the input variables follow a pattern of the form `t_outputName_start`, `t_outputName_stop`. For instance, *t_graph_start*, *t_graph_stop*, *t_log_start*, *t_log_stop*, *t_spec_start*, *t_spec_stop*, etc.

---

2. **User-defined interval between two outputs, given in number of time steps**. Again the last output is performed at the end of the run and a compensation may take place at the beginning.

Note: These input variables are usually named with a pattern that follows `n_outputName_step`, for instance, *n_graph_step*, *n_rst_step*, *n_spec_step*, *n_log_step*, *n_movie_step*, etc.

---

3. **Defined time interval between two outputs**.

Note: These input variables are usually named with a pattern that follows `dt_outputName`, for instance, *dt_graph*, *dt_rst*, *dt_spec*, *dt_log*, *dt_movie*, etc.

---

4. **User-defined times for output**. By default 5000 different times can be defined for each output type. This can be increased by increasing n_time_hits in the file `output_data.f90`. While the first three possibilities can only be used alternatively, the fourth one can be employed in addition to one of the two others.

Note: These input variables are usually named with a pattern that follows `t_outputName`, for instance,

t_graph, *t_rst*, *t_spec*, *t_log*, *t_movie*, etc.

An important parameter in this context is *l_true_time*. If this is set to `.true.`, the time steps of the program are modified to meet a desired output time. This forces a recalculation of the inversion matricies and therefore requires some additional computing time. When `l_true_time=.false.`, the values at the timestep closest to the desired output time are chosen. Since the timesteps are generally small, this option suffices for most applications.

- **l_true_time** (default *l_true_time=.false.*) is a logical. It causes the code to change time steps to exactly meet the requested output times.

The different possible outputs control parameters are then extensively described in the following pages:

Possible outputs

1. *Control standard/common outputs*
2. *CMB and radial coefficients*
3. *Storage of potentials in spectral space*
4. *Torsional oscillations diagnostics*
5. *Additional possible diagnostics*

6.6.1 Standard time-series outputs

The **log** outputs controls the output of all the default time series of the file: kinetic and magnetic energies (*e_kin.TAG*, *e_mag_oc.TAG* and *e_mag_ic.TAG* files), dipole information (*dipole.TAG* file), rotation (*rot.TAG*) parameters (*par.TAG*) and various additional diagnostics (*misc.TAG*):

- **n_log_step** (default *n_log_step=50*) is an integer. This is the number of timesteps between two log outputs.

  Warning: Be careful: when using too small *n_log_step*, the disk access will dramatically increases, thus decreasing the code performance.

- **n_logs** (default *n_logs=0*) is an integer. This is the number of log-information sets to be written.
- **t_log** (default *t_log=-1.0 -1.0 ...*) is real array, which contains the times when log outputs are requested.
- **dt_log** (default *dt_log=0.0*) is a real, which defines the time interval between log outputs.
- **t_log_start** (default *t_log_start=0.0*) is a real, which defines the time to start writing log outputs.
- **t_log_stop** (default *t_log_stop=0.0*) is a real, which defines the time to stop writing log outputs.

6.6.2 Restart files

The **rst** outputs controls the output of restart files (*rst_t_#.TAG*) (i.e. check points in time from which the code could be restarted):

- **n_rst_step** (default *n_rst_step=0*) is an integer. This is the number of timesteps between two restart files.
- **n_rsts** (default *n_rsts=1*) is an integer. This is the number of restart files to be written.
- **t_rst** (default *t_rst=-1.0 -1.0 ...*) is real array, which contains the times when restart files are requested.
- **dt_rst** (default *dt_rst=0.0*) is a real, which defines the time interval between restart files.
- **t_rst_start** (default *t_rst_start=0.0*) is a real, which defines the time to start writing restart files.

- **t_rst_stop** (default *t_rst_stop=0.0*) is a real, which defines the time to stop writing restart files.
- **n_stores** (default *n_stores=0*) is an integer. This is another way of requesting a certain number of restart files. However, instead of creating each time a new restart file, if `n_stores > n_rsts` the restart file is overwritten, which can possibly help saving some disk space.

Warning: The `rst` files can become quite big and writting them too frequently will slow down the code. Except for very special use, the default set up should be sufficient.

6.6.3 Graphic files

The **graph** outputs controls the output of graphic files (*G_#.TAG*) which contain a snapshot the entropy, the velocity field and the magnetic fields:

- **n_graph_step** (default *n_graph_step=0*) is an integer. This is the number of timesteps between two graphic files.
- **n_graphs** (default *n_graphs=1*) is an integer. This is the number of graphic files to be written.
- **t_graph** (default *t_graph=-1.0 -1.0 ...*) is real array, which contains the times when graphic files are requested.
- **dt_graph** (default *dt_graph=0.0*) is a real, which defines the time interval between graphic files.
- **t_graph_start** (default *t_graph_start=0.0*) is a real, which defines the time to start writing graphic files.
- **t_graph_stop** (default *t_graph_stop=0.0*) is a real, which defines the time to stop writing graphic files.

6.6.4 Spectra

The **spec** outputs controls the output of spectra: kinetic energy spectra (*kin_spec_#.TAG*), magnetic energy spectra (*mag_spec_#.TAG*) and thermal spectra (*T_spec_#.TAG*):

- **n_spec_step** (default *n_spec_step=0*) is an integer. This is the number of timesteps between two spectra.
- **n_specs** (default *n_specs=0*) is an integer. This is the number of spectra to be written.
- **t_spec** (default *t_spec=-1.0 -1.0 ...*) is real array, which contains the times when spectra are requested.
- **dt_spec** (default *dt_spec=0.0*) is a real, which defines the time interval between spectra.
- **t_spec_start** (default *t_spec_start=0.0*) is a real, which defines the time to start writing spectra.
- **t_spec_stop** (default *t_spec_stop=0.0*) is a real, which defines the time to stop writing spectra.

6.6.5 Movie files

The **movie** outputs controls the output of movie files (**_mov.TAG*).

Specific inputs

- **l_movie** (default *l_movie=.false.*) is a logical. It needs to be turned on to get movie computed.

  Several movie-files can be produced during a run (it is now limited to 30 by the variable‘‘n_movies_max‘‘ in the module `movie`). The movies are defined by a keyword determining the fields to be plotted and an expression

that determines the nature of movie (r-slice, θ-slice, ϕ-slice, etc.). The code searches this information in a character string provided for each movie. These strings are elements of the array *movie*:

- **movie** (default `movie=' ', ' ', ...`) is a character string array. It contains the description of the movies one wants to compute.

  For example, to invoke a movie(file) that shows (stores) the radial magnetic component of the magnetic field at the CMB, you have to provide the line

```
movie(1)="Br CMB",
```

in the *&output* namelist. Here, `Br` is the keyword for the radial component of the magnetic field and `CMB` is the expression that defines the movie surface. If, in addition, a movie of the temperature field at the meridional slice `phi=0` and a movie of the z-vorticity in the equatorial plane are desired, the following line have to be added:

```
movie(2)="Temp phi=0",
movie(3)="Vortz eq",
```

Note that the code does **not interpret spaces and ignores additional characters** that do not form a keyword or a surface definition. Thus, for example `Br` or `B r` or `Bradial` are all interpreted as the same keyword. Furthermore, the interpretation is **not case-sensitive**. The following table gives the possible keywords for movie calculations and their corresponding physical meaning:

Keyword	Fields stored in movie file
Br[radial]	Radial component of the magnetic field B_r.
Bt[heta]	Latitudinal component of the magnetic field B_θ.
Bp[hi]	Azimuthal component of the magnetic field B_ϕ.
Bh[orizontal]	The two horizontal components of the magnetic field.
Bs	Cylindrically radial component of the magnetic field B_s.
Ba[ll]	All magnetic field components.
Fieldline[s] or FL	Axisymmetric poloidal field lines in a meridional cut.
AX[ISYMMETRIC] B or AB	Axisymmetric phi component of the magnetic field for $\phi = cst$.
Vr[adial]	Radial component of the velocity field u_r.
Vt[heta]	Latitudinal component of the velocity field u_θ.
Vp[hi]	Azimuthal component of the velocity field u_ϕ.
Vh[orizontal]	Horizontal velocity field, two components depending on the surface.
Va[ll]	All velocity field components.
Streamline[s] or SL	Field lines of axisymmetric poloidal field for $\phi = cst$.
AX[ISYMMETRIC] V or AV	Axisymmetric component of the velocity field for $\phi = cst$.
Vz	Vertical component of the velocity at the equator + vertical component of the vorticity at the equator (closest point to equator).
Voz	Vertical component of the vorticity ω_z.
Vor	Radial component of the vorticity ω_r.
Vop	Azimuthal component of vorticity ω_ϕ
Tem[perature] or Entropy	Temperature/Entropy
Entropy (or Tem[perature]) AX[ISYMMETRIC] or AT	Axisymmetric temperature/entropy field for $\phi = cst$.
Heat t[ransport]	Radial advection of temperature $u_r \frac{\partial s}{\partial r}$
HEATF AX[iSYMMETRIC]	Conducting heat flux $\partial s/\partial r$

Continued on next page

Table 6.1 – continued from previous page

Keyword	Fields stored in movie file
FL Pro	Axisymmetric field line stretching.
FL Adv	Axisymmetric field line advection.
FL Dif	Axisymmetric field line diffusion.
AB Pro	Toroidal axisymmetric field production.
AB Dif	Toroidal axisymmetric field diffusion.
Br Pro	Production of radial magnetic field B_r.
Br Adv	Advection of radial magnetic field B_r.
Br Dif	Diffusion of radial magnetic field B_r.
Jr	Radial component of the current j_r.
Jr Pro	Production of radial current + Ω-effect.
Jr Adv	Advection of the radial component of the current j_r.
Jr Dif	Diffusion of the radial component of the current j_r.
Bz Pol	Poloidal part of vertical component of the magnetic field B_z.
Bz Pol Pro	Production of the poloidal part of the vertical component of the magnetic field B_z.
Bz Pol Adv	Advection of the poloidal part of the vertical component of the magnetic field B_z.
Bz Pol Dif	Diffusion of the poloidal part of the vertical component of the magnetic field B_z.
Jz Tor	Toroidal part of the vertical component of the current (j_z).
Jz Tor Pro	Production of the toroidal part of the vertical component of the current j_z.
Jz Tor Adv	Advection of the toroidal part of the vertical component of the current j_z.
Jz Tor Dif	Diffusion of the toroidal part of the vertical component of the current j_z.
Bp Tor	Toroidal part of the azimuthal component of the magnetic field B_ϕ.
Bp Tor Pro	Production of the toroidal part of the azimuthal component of the magnetic field B_ϕ.
Bp Tor Adv	Advection of the toroidal part of the azimuthal component of the magnetic field B_ϕ.
Bp Tor Dif	Diffusion of the toroidal part of the azimuthal component of the magnetic field B_ϕ.
HEL[ICITY]	Kinetic helicity $\mathcal{H} = \boldsymbol{u} \cdot (\boldsymbol{\nabla} \times \boldsymbol{u})$
AX[ISYMMETRIC HELICITY] or AHEL	Axisymmetric component of the kinetic helicity.
Bt Tor	Toroidal component of the latitudinal component of the magnetic field B_θ.
Pot Tor	Toroidal potential.
Pol Fieldlines	Poloidal fieldlines.
Br Shear	Azimuthal shear of the radial component of the magnetic field B_r
Lorentz[force] or LF	Lorentz force (only ϕ-component).
Br Inv	Inverse field apperance at CMB.

The following table gives the possible surface expression for movie calculations and their corresponding physical meaning:

Surface expression	Definition
CMB	Core-mantle boundary
Surface	Earth surface
EQ[uatot]	Equatorial plane
r=radius	Radial cut at r=radius with radius given in units of the outer core radius.
theta=colat	Latitudinal cut at theta=colat given in degrees
phi=phiSlice	Azimuthal cut ath phi=phiSlice given in degrees.
AX[isymmetric]	Axisymmetric quantity in an azimuthal plane
3D	3D array

Here is an additional example of the possible combinations to build your desired `movie` files.

```
l_movie  = .true.,
movie(1) = "Br CMB",
movie(2) = "Vr EQ",
movie(3) = "Vortr r=0.8",
movie(4) = "Bp theta=45",
movie(5) = "Vp phi=10",
movie(6) = "entropy AX",
movie(7) = "vr 3D",
```

Standard inputs

- **n_movie_step** (default *n_movie_step=0*) is an integer. This is the number of timesteps between two movie outputs.
- **n_movies** (default *n_movies=1*) is an integer. This is the number of movie outputs to be written.
- **t_movie** (default *t_movie=-1.0 -1.0 ...*) is real array, which contains the times when movie outputs are requested.
- **dt_movie** (default *dt_movie=0.0*) is a real, which defines the time interval between movie outputs.
- **t_movie_start** (default *t_movie_start=0.0*) is a real, which defines the time to start writing movie outputs.
- **t_movie_stop** (default *t_movie_stop=0.0*) is a real, which defines the time to stop writing movie outputs.

6.6.6 Field Averages

The code can perform on-the-fly time-averaging of entropy, velocity field and magnetic field. Respective graphic output and spectra are written into the corresponding files (with *G_ave.TAG*, *kin_spec_ave.TAG*, *mag_spec_ave.TAG*). The time-averaged energies are written into the *log.TAG* file.

- **l_average** (default *l_average=.false.*) is a logical, which enables the time-averaging of fields when set to `.true.`.

> **Warning:** Time-averaging has a large memory imprint as it requires the storage of 3-D arrays. Be careful, when using large truncations.

6.6.7 Poloidal magnetic field potential at CMB

The **cmb** outputs controls the output of poloidal field potential coefficients at the CMB $b_{\ell m}(r = r_o)$: *B_coeff_cmb.TAG* up to a maximum spherical harmonic degree `l_max_cmb`.

Note: This calculation is **only** enabled when `l_cmb_field=.true.` or when `l_dt_cmb_field=.true.`.

Specific inputs

- **l_cmb_field** (default *l_cmb_field=.false.*) is a logical. It needs to be turned on to get cmb files computed.
- **l_dt_cmb_field** (default *l_dt_cmb_field=.false.*) is a logical. When set to .true., it allows the calculation of the secular variation of the magnetic field at the CMB.
- **l_max_cmb** (default *l_max_cmb=14*) is an integer. This is the maximum spherical harmonic degree ℓ stored in *B_coeff_cmb.TAG*, i.e. only $\ell \leq \ell_{maxcmb}$ are stored. For example, the following input parameter means that the *B_coeff_cmb.TAG* file is stored up to a spherical harmonic degree of ℓ:

```
l_cmb_field = .true.,
l_max_cmb   = 20,
```

Standard inputs

- **n_cmb_step** (default *n_cmb_step=0*) is an integer. This is the number of timesteps between two cmb outputs.
- **n_cmbs** (default *n_cmbs=0*) is an integer. This is the number of cmb outputs to be written.
- **t_cmb** (default *t_cmb=-1.0 -1.0 ...*) is real array, which contains the times when cmb outputs are requested.
- **dt_cmb** (default *dt_cmb=0.0*) is a real, which defines the time interval between cmb outputs.
- **t_cmb_start** (default *t_cmb_start=0.0*) is a real, which defines the time to start writing cmb outputs.
- **t_cmb_stop** (default *t_cmb_stop=0.0*) is a real, which defines the time to stop writing cmb outputs.

6.6.8 Poloidal and toroidal potentials at several depths

The coeff_r# outputs controls the output of the poloidal and toroidal potential coefficients at several depths up to a maximum spherical harmonic degree l_max_r. The files *B_coeff_r#.TAG* and *V_coeff_r#.TAG* are written when l_r_field=.true.. The file *T_coeff_r#.TAG* is written when l_r_fieldT=.true..

Note: This calculation is **only** enabled when l_r_field=.true. or when l_r_fieldT=.true..

Specific inputs

- **l_r_field** (default *l_r_field=.false.*) is a logical. It needs to be turned on to get r_field files computed.
- **l_r_fieldT** (default *l_r_fieldT=.false.*) is a logical. When set to .true., the thermal field is also stored in a file named *T_coeff_r*.TAG*.
- **l_max_r** (default *l_max_r=l_max*) is an integer. This is the maximum spherical harmonic degree ℓ stored in the r_field file, i.e. only $\ell \leq \ell_{maxcmb}$ are stored.

There are two ways to specify the radial grid points where you want to store the [B|V|T]_coeff_r#.TAG files. You can specify a stepping n_r_step: in that case **5** coeff_r#.TAG files will be stored at 5 different radial levels every n_r_step grid point:

```
l_r_field = .true.,
n_r_step  = 6,
l_max_r   = 30,
```

This will produces 5 files that contain the poloidal and toroidal potentials up to spherical harmonic degree $\ell = 30$:

- `[B|V|T]_coeff_r1.TAG` corresponds to the radial grid point with the index `nR=6`.
- `[B|V|T]_coeff_r2.TAG` to `nR=12`.
- `[B|V|T]_coeff_r3.TAG` to `nR=18`.
- `[B|V|T]_coeff_r4.TAG` to `nR=24`.
- `[B|V|T]_coeff_r5.TAG` to `nR=30`.

- **n_r_step** (default *n_r_step=2*) is an integer. This specifies the stepping between two consecutive `[B|V|T]_coeff_r#.TAG` files.

Alternatively, the input array `n_r_array` can be used to specify the radial grid points you exactly want to store:

```
l_r_field = .true.,
n_r_array = 8, 24, 47,
l_max_r   = 10,
```

This will produces 3 files that contain the poloidal and toroidal potentials up to spherical harmonic degree $\ell = 10$:

- `[B|V|T]_coeff_r1.TAG` corresponds to the radial grid point with the index `nR=8`.
- `[B|V|T]_coeff_r2.TAG` to `nR=24`.
- `[B|V|T]_coeff_r3.TAG` to `nR=47`.

- **n_r_array** (default *n_r_array=0 0 0 ...*) a an integer array. You can specify the radial grid points (starting from `n_r_cmb=1`) where you want to store the coefficients.

Standard inputs

- **n_r_field_step** (default *n_r_field_step=0*) is an integer. This is the number of timesteps between two `r_field` outputs.
- **n_r_fields** (default *n_r_fields=0*) is an integer. This is the number of `r_field` outputs to be written.
- **t_r_field** (default *t_r_field=-1.0 -1.0 ...*) is real array, which contains the times when `r_field` outputs are requested.
- **dt_r_field** (default *dt_r_field=0.0*) is a real, which defines the time interval between `r_field` outputs.
- **t_r_field_start** (default *t_r_field_start=0.0*) is a real, which defines the time to start writing `r_field` outputs.
- **t_r_field_stop** (default *t_r_field_stop=0.0*) is a real, which defines the time to stop writing `r_field` outputs.

6.6.9 Poloidal and toroidal potentials in spectral and Chebyshev space

The **[V|B|T]pot** outputs controls the output of potential files (*Vpot_#.TAG*, *Bpot_#.TAG* and *Tpot_#.TAG*). These are files that contain the poloidal and toroidal flow and magnetic field potentials (and entropy/temperature) written in spectral and Chebyshev spaces (for instance `w(lm_max, n_cheb_max)`). These files can be pretty useful since

they can be possibly used to reconstruct any quantity in the spectral space or in the physical space you may be interested in.

Specific inputs

They are two ways to store those files. The first option is to use *l_storePot=.true.* and the corresponding time control parameters (*n_pot_step*, *t_pot*, *n_pots*, etc.). In that case the three files *Vpot_#.TAG*, *Bpot_#.TAG* and *Tpot_#.TAG* will be stored. The following example will create new *Vpot_#.TAG*, *Bpot_#.TAG* and *Tpot_#.TAG* files every 1000 time steps:

```
    l_storePot = .true.,
    n_pot_step = 1000,
```

- **l_storePot** (default *l_storePot=.false.*) is a logical. It needs to be turned on to store all the potentials in three different files: *Vpot_#.TAG*, *Bpot_#.TAG* and *Tpot_#.TAG*.

The second option is control separately the writing of the three files using the three logicals *l_storeVpot*, *l_storeBpot* and *l_storeTpot* and their corresponding time control parameters. The following example wrill create a new *Vpot_#.TAG* file every 1000 time steps and a new *Bpot_#.TAG* file every 3000 time steps (no *Tpot_#.TAG* files are stored in that case):

```
    l_storeVpot = .true.,
    n_Vpot_step = 1000,
    l_storeBpot = .true.,
    n_Bpot_step = 3000,
    l_storeTpot = .false.,
```

- **l_storeVpot** (default *l_storeVpot=.false.*) is a logical. It needs to be turned on to store the flow poloidal and toroidal potentials. It then writes the *Vpot_#.TAG* file.
- **l_storeBpot** (default *l_storeBpot=.false.*) is a logical. It needs to be turned on to store the magnetic field poloidal and toroidal potentials. It then writes the *Bpot_#.TAG* file.
- **l_storeTpot** (default *l_storeTpot=.false.*) is a logical. It needs to be turned on to store the entropy. It then writes the *Tpot_#.TAG* file.

Standard inputs

- **n_pot_step** (default *n_pot_step=0*) is an integer. This is the number of timesteps between two `[V|B|P]pot` outputs.
- **n_pots** (default *n_pots=1*) is an integer. This is the number of `[V|B|P]pot` outputs to be written.
- **t_pot** (default *t_pot=-1.0 -1.0 ...*) is real array, which contains the times when `[V|B|P]pot` outputs are requested.
- **dt_pot** (default *dt_pot=0.0*) is a real, which defines the time interval between two `[V|B|P]pot` outputs.
- **t_pot_start** (default *t_pot_start=0.0*) is a real, which defines the time to start writing `[V|B|P]pot` outputs.
- **t_pot_stop** (default *t_pot_stop=0.0*) is a real, which defines the time to stop writing `[V|B|P]pot` outputs.
- **n_Vpot_step** (default *n_Vpot_step=0*) is an integer. This is the number of timesteps between two `Vpot` outputs.
- **n_Vpots** (default *n_Vpots=1*) is an integer. This is the number of `Vpot` outputs to be written.

- **t_Vpot** (default *t_Vpot=-1.0 -1.0 ...*) is real array, which contains the times when Vpot outputs are requested.
- **dt_Vpot** (default *dt_Vpot=0.0*) is a real, which defines the time interval between Vpot outputs.
- **t_Vpot_start** (default *t_Vpot_start=0.0*) is a real, which defines the time to start writing Vpot outputs.
- **t_Vpot_stop** (default *t_Vpot_stop=0.0*) is a real, which defines the time to stop writing Vpot outputs.
- **n_Bpot_step** (default *n_Bpot_step=0*) is an integer. This is the number of timesteps between two Bpot outputs.
- **n_Bpots** (default *n_Bpots=1*) is an integer. This is the number of Bpot outputs to be written.
- **t_Bpot** (default *t_Bpot=-1.0 -1.0 ...*) is real array, which contains the times when Bpot outputs are requested.
- **dt_Bpot** (default *dt_Bpot=0.0*) is a real, which defines the time interval between Bpot outputs.
- **t_Bpot_start** (default *t_Bpot_start=0.0*) is a real, which defines the time to start writing Bpot outputs.
- **t_Bpot_stop** (default *t_Bpot_stop=0.0*) is a real, which defines the time to stop writing Bpot outputs.
- **n_Tpot_step** (default *n_Tpot_step=0*) is an integer. This is the number of timesteps between two Tpot outputs.
- **n_Tpots** (default *n_Tpots=1*) is an integer. This is the number of Tpot outputs to be written.
- **t_Tpot** (default *t_Tpot=-1.0 -1.0 ...*) is real array, which contains the times when Tpot outputs are requested.
- **dt_Tpot** (default *dt_Tpot=0.0*) is a real, which defines the time interval between Tpot outputs.
- **t_Tpot_start** (default *t_Tpot_start=0.0*) is a real, which defines the time to start writing Tpot outputs.
- **t_Tpot_stop** (default *t_Tpot_stop=0.0*) is a real, which defines the time to stop writing Tpot outputs.

6.6.10 Torsional oscillations (TO)

Specific inputs

- **l_TO** (default *l_TO=.false.*) is a logical. It needs to be turned on to compute the torsional oscillations diagnostics (TO) computed.
- **l_TOmovie** (default *l_TOmovie=.false*) is a logical. It needs to be turned on to store the *TO_movie.TAG* files.
- **sDens** (default *sDens=1.0*) is a float. It gives the relative point density of the cylindrical grid (in the radial direction).
- **zDens** (default *zDens=1.0*) is a float. It gives the relative point density of the cylindrical grid (in the vertical direction).

Standard inputs

- **n_TO_step** (default *n_TO_step=0*) is an integer. This is the number of timesteps between two TO outputs.
- **n_TOs** (default *n_TOs=1*) is an integer. This is the number of TO outputs to be written.
- **t_TO** (default *t_TO=-1.0 -1.0 ...*) is real array, which contains the times when TO outputs are requested.
- **dt_TO** (default *dt_TO=0.0*) is a real, which defines the time interval between TO outputs.

- **t_TO_start** (default *t_TO_start=0.0*) is a real, which defines the time to start writing TO outputs.
- **t_TO_stop** (default *t_TO_stop=0.0*) is a real, which defines the time to stop writing TO outputs.
- **n_TOZ_step** (default *n_TOZ_step=0*) is an integer. This is the number of timesteps between two TO outputs.
- **n_TOZs** (default *n_TOZs=1*) is an integer. This is the number of TO outputs to be written.
- **t_TOZ** (default *t_TOZ=-1.0 -1.0 ...*) is real array, which contains the times when TO outputs are requested.
- **dt_TOZ** (default *dt_TOZ=0.0*) is a real, which defines the time interval between TO outputs.
- **t_TOZ_start** (default *t_TOZ_start=0.0*) is a real, which defines the time to start writing TO outputs.
- **t_TOZ_stop** (default *t_TOZ_stop=0.0*) is a real, which defines the time to stop writing TO outputs.
- **n_TOmovie_step** (default *n_TOmovie_step=0*) is an integer. This is the number of timesteps between two TO_mov outputs.
- **n_TOmovie_frames** (default *n_TOmovies=1*) is an integer. This is the number of TO_mov outputs to be written.
- **t_TOmovie** (default *t_TOmovie=-1.0 -1.0 ...*) is real array, which contains the times when TO_mov outputs are requested.
- **dt_TOmovie** (default dt_TOmovie=0.0) is a real, which defines the time interval between TO_mov outputs.
- **t_TOmovie_start** (default *t_TOmovie_start=0.0*) is a real, which defines the time to start writing TO_mov outputs.
- **t_TOmovie_stop** (default *t_TOmovie_stop=0.0*) is a real, which defines the time to stop writing TO_mov outputs.

6.6.11 RMS force balance

The code can compute the RMS contributions of the different forces that contribute to the Navier-Stokes equation and the the different terms that enter the induction equation.

- **l_RMS** (default *l_RMS=.false.*) is a logical, which enables the calculation of RMS force balance, when set to .true.. The outputs are stored in *dtVrms.TAG* and *dtBrms.TAG*.
- **rCut** (default *rCut=0.0*) is a float. This is the thickness of the layer which is left out at both boundaries for the RMS calculation. rCut=0.075 actually means that 7.5% below the CMB and above the ICB are disregarded in the force balance calculation.
- **rDea** (default *rDea=0.0*) is a float. This controls the dealiasing in RMS calculations. rDea=0.1 means that the highest 10% of the Chebyshev modes are set to zero.

6.6.12 Additional possible diagnostics

Geostrophy

- **l_par** (default *l_par=.false.*) is a logical. When set to .true., this logical enables additional calculations (for instance the degree of geostrophy). The details of these caclulations can be found in the subroutine getEgeos in the Egeos.f90 file. These quantities are then stored in the columns 10-16 of the *misc.TAG* file.
- **l_corrMov** (default *l_corrMov=.false.*) is a logical. When set to .true., this logical enables the calculation of a movie file that stores North/South correlation in the CVorz_mov.TAG file.

Helicity

- **l_hel** (default *l_hel=.false.*) is a logical. When set to .true., this logical enables the calculation of helicity (RMS, northern and southern hemisphere, etc.). The outputs are stored in the columns 6-9 of the *misc.TAG* file.

Power budget

- **l_power** (default *l_power.false.*) is a logical. When set to .true., this logical enables the calculation of input and output power (buoyancy, viscous and ohmic dissipations, torques). The time series are stored in *power.TAG* and *dtE.TAG* and the time-averaged radial profiles in *powerR.TAG*.

Angular momentum

- **l_AM** (default *l_AM=.false.*) is a logical. When set to .true., this logical enables the calculation of angular momentum. The time series are stored in *AM.TAG*.

Drift rates

- **l_drift** (default *l_drift=.false.*) is a logical. When set to .true., this logical enables the storage of some selected coefficients to allow the calculation of the drift rate. The time series are stored in *drift[V\B][DQ].TAG*.

Inertial modes

- **l_iner** (default *l_iner=.false.*) is a logical. When set to .true., this logical enables the storage of some selected $w(\ell, m)$ at mid-shell (stored in *inerP.TAG*) and $z(\ell, m)$ at mid-shell (stored in *inerT.TAG*). Those files can be further used to identify inertial modes.

Radial spectra

- **l_rMagSpec** (default *l_rMagSpec=.false*) is a logical. When set to .true., the magnetic spectra for the first 6 spherical harmonic degree ℓ for all radii are stored at times of log ouputs. This produces the unformatted fortran files *rBrSpec.TAG* and *rBpSpec.TAG*.
- **l_DTrMagSpec** (default *l_DTrMagSpec=.false*) is a logical. When set to .true., the magnetic spectra of the magnetic field production terms for the first 6 spherical harmonic degree ℓ for all radii are stored at times of log ouputs. This produces the unformatted fortran files rBrProSpec.TAG, rBrAdvSpec.TAG, rBrDifSpec.TAG, rBrDynSpec.TAG, rBpProSpec.TAG, rBpAdvSpec.TAG, rBpDifSpec.TAG and rBpDynSpec.TAG. All those files have exactly the same format as the *rBrSpec.TAG*.

Heat transport

- **l_fluxProfs** (default *l_fluxProfs=.false.*) is a logical. When set to .true., this logical enables the calculation of time-averaged radial heat flux profiles (conductive flux, convective flux, kinetic flux, viscous flux, Poynting flux and resistive flux). The time-averaged radial profiles are stored in the *fluxesR.TAG* file.

Boundary layer analysis

- **l_viscBcCalc** (default *l_viscBcCalc=.false.*) is a logical. When set to .true., this logical enables the calculation of time-averaged radial profiles that can be further use to determine the viscous and thermal boundary layer thicknesses: temperature, temperature variance, horizontal velocity, etc. The time-averaged radial profiles are stored in the *bLayersR.TAG* file.

Parallel/perpendicular decomposition

- **l_perpPar** (default *l_perpPar=.false.*) is a logical. When set to .true., this logical enables the decomposition of kinetic energy into components parallel and perpendicular to the rotation axis. The time series are stored in *perpPar.TAG* and the time-averaged radial profiles in *perpParR.TAG*.

Potential vorticity

- **l_PV** (default *l_PV=.false.*) is a logical. When set to .true., this logical enables some potential vorticity diagnostics. At the end of the run, the results are stored in the the files PVZ.TAG and Vcy.TAG.

Pressure

- **l_PressGraph** (default *l_PressGraph=.true.*) is a logical. When set to .true., this logical enables the storage of pressure in the *graphic files*.

6.6.13 Generic options

- **l_save_out** (default *l_save_out=.false.*) is a logical. When set to .true., the diagnostic files will be safely opened and closed before and after any outputs. When set to .false., the diagnostic files will be opened before the first iteration timestep and close at the end of the run. This may cost some computing time, but guarantees that only minimal information is lost in case of a crash.
- **lVerbose** (default *lVerbose=.false.*) is a logical. When set to .true., the code displays a lot of debugging informations.

  Warning: Never set lVerbose to .true. for a production run!

- **runid** (default, *runid="MAGIC default run"*) is a character string. This can be used to briefly describe your run. This information is then for instance stored in the header of the graphic files.

6.7 Mantle and Inner Core Namelists

6.7.1 Mantle Namelist

This namelist defines mantle properties

- **conductance_ma** (default *conductance_ma=0.0*) is a real that defines the conductance (dimensionless) of the mantle.
- **nRotMa** (default *nRotMa=0*) is an integer that defines the rotation of the mantle:

nRotMa=-1	Mantle rotates with prescribed rate (see `omega_ma1` and `omega_ma2` below)
nRotMa=0	Fixed, non-rotating mantle
nRotMa=1	Mantle rotates according to torques

- **rho_ratio_ma** (default *rho_ratio_ma=1*) is a real which gives the density of the mantle in terms of that of the outer core.
- **omega_ma1** (default *omega_ma1=0.0*) is a real which defines a mantle rotation rate (used when `nRotMa=0`).
- **omegaOsz_ma1** (default *omegaOsz_ma1=0.0*) is a real which prescribes the oscillation frequency of the mantle rotation rate. In this case, `omega_ma1` is the amplitude of the oscillation.
- **tShift_ma1** (default *tShift_ma1=0.0*) is a real which defines the time shift of the mantle rotation rate `omega_ma1`.
- **omega_ma2** (default *omega_ma2=0.0*) is a real which defines a second mantle rotation rate.
- **omegaOsz_ma2** (default *omegaOsz_ma2=0.0*) is a real which defines the oscillation frequency of the second mantle rotation rate `omega_ma2`.
- **tShift_ma2** (default *tShift_ma2=0.0*) is a real which defines the time shift for `omega_ma2`.

The resultant prescribed mantle rotation rate is computed as:

```
omega_ma = omega_ma1*cos(omegaOsz_ma1*(time+tShift_ma1)) + &
           omega_ma2*cos(omegaOsz_ma2*(time+tShift_ma2))
```

6.7.2 Inner Core Namelist

This namelist defines properties of the inner core

- **sigma_ratio** (default *sigma_ratio=0.0*) is a real that defines the conductivity of the inner core with respect to the value of the outer core. `sigma_ratio=0` thus corresponds to a non-conducting inner core.
- **nRotIc** (default *nRotIc=0*) is an integer that defines the rotation of the inner core. Behaves the same way as *nRotMa* (above).
- **rho_ratio_ic** (default *rho_ratio_ic=1.0*) is a real which defines the density of the inner core in terms of that of the outer core.
- **BIC** (default *BIC=0.0*) is a real which gives the imposed dipole field strength at the Inner Core Boundary. Having `BIC > 0` implies that the inner core acts as a dipole magnet - as implemented in the DTS experiment at Grenoble, France.
- **Variables prescribing rotation rate of inner core** The following variables are used to prescribe rotation rate of the inner core. They behave in the same way as the corresponding variables for the mantle. They are usd only when `nRotIC=0`.
  - **omega_ic1** (default *omega_ic1=0.0*)
  - **omegaOsz_ic1** (default *omegaOsz_ic1=0.0*)
  - **tShift_ic1** (default *tShift_ic1=0.0*)
  - **omega_ic2** (default *omega_ic2=0.0*)
  - **omegaOsz_ic2** (default *omegaOsz_ic2=0.0*)
  - **tShift_ic2** (default *tShift_ic2=0.0*)

As with the mantle, the resultant prescribed rotation rate for the inner core is computed as:

```
omega_ic = omega_ic1*cos(omegaOsz_ic1*(time+tShift_ic1)) + &
           omega_ic2*cos(omegaOsz_ic2*(time+tShift_ic2))
```

CHAPTER

SEVEN

INTERACTIVE COMMUNICATION WITH THE CODE USING `SIGNAL.TAG`

It is possible to interactively communicate with the MagIC code **during a run**, using a file which is systematically created when the simulation starts, called **signal.TAG**. By default, this file contains only the word `NOT` and does nothing to the simulation. Replacing `NOT` by one of the following allowed keywords will have some influence on the outputs or possibly force the code to terminate its execution:

- **END**: Changing the word `NOT` to `END` will cause the code to finish after the current time step and write all the outputs as if it was programmed to finish at that time from the start. This will thus normally produce the *rst_end.TAG* file that will possibly allow you to continue this run later at your convenience.
- **GRA**: Changing the word `NOT` to `GRA` will cause the code to produce a graphic ouptut file *G_#.TAG*. The keyword will be automatically restored to `NOT` once the graphic file has been produced.
- **RST**: Changing the word `NOT` to `RST` will cause the code to produce a restart file *rst_t#.TAG*. The keyword will then be restored to `NOT` once the restart file has been written.
- **SPE**: Changing the word `NOT` to `SPE` will cause the code to produce spectra *kin_spec_#.TAG* (and possibly *mag_spec_#.TAG* and *T_spec_#.TAG* <*secTSpecFile*> depending if the run is magnetic or not, or if it solves a temperature/entropy equation). Once the spectra files have been written, the keyword will be automatically replaced by `NOT`.

Note: Those keywords are **case-insensitive**.

Instead of editing the file with your favorite editor to specify the requested keyword, we recommand using instead the shell command `echo` to avoid some possible crash during the code execution when writing into the `signal.TAG` file. For instance, if you want a *graphic output file*, just use the following command (adapted to your current *TAG*):

```
$ echo GRA > signal.TAG
```

CHAPTER

EIGHT

OUTPUT FILES

While some information of a run is written into `STDOUT` to monitor its progress, most outputs are printed into dedicated files identified by the chosen *TAG* extension. These files can be parsed and analysed using the *python classes*. The following pages describe the content and the structure of the different type of output files:

1. Most of the information found in `STDOUT` is also written to the **log-file** called *log.TAG*. In addition, this file contains all input parameters, truncation, information on other output files, and some results like the time averaged energies (when *l_average=.true.*).
2. There are several ascii files that contain the **time-evolution of integrated quantities** (energies, heat fluxes, rotation rate, Reynolds numbers, etc.) that are systematically produced:
   - Kinetic energies: *e_kin.TAG*,
   - Magnetic energies: *e_mag_oc.TAG* and *e_mag_ic.TAG*,
   - Rotation rates: *rot.TAG*,
   - Informations about the dipolar component of the magnetic field: *dipole.TAG*,
   - Diagnostic parameters (Reynolds, Elsasser, etc.): *par.TAG*,
   - Additional diagnostics (heat fluxes, Nusselt numbers, etc.): *misc.TAG*.
3. There are **additional conditional time series** that contain the time-evolution of other physical quantities that depend on the chosen *input parameters*:
   - Angular momentum balance: *AM.TAG*,
   - Power budget: *power.TAG* and *dtE.TAG*,
   - Square velocities: *u_square.TAG*,
   - Drift rates: *drift[V\B][D\Q].TAG* and *iner[P\T].TAG*,
   - Torques: *SR[IC\MA].TAG*,
   - RMS calculations of the force balances: *dtVrms.TAG* and *dtBrms.TAG*,
   - Kinetic energies perpendicular and parallel to the rotation axis: *perpPar.TAG*.
4. **Time-averaged radial profiles**:
   - Kinetic energies: *eKinR.TAG*,
   - Magnetic energies: *eMagR.TAG*,
   - Diagnostic quantities: *parR.TAG*,
   - Power budget: *powerR.TAG*,
   - Heat fluxes: *fluxesR.TAG*,

- Temperature and horizontal velocities: *bLayersR.TAG*,
- Kinetic energies perpendicular and parallel to the rotation axis: *perpParR.TAG*.

5. **Radial profiles of the transport properties** of the reference state (those files will only be produced when the appropriate input option is chosen):
   - Temperature, density and gravity: *anel.TAG*,
   - Electrical conductivity: *varCond.TAG*,
   - Thermal conductivity: *varDiff.TAG*,
   - Kinematic viscosity: *varVisc.TAG*,
   - Mapping of the Chebyshev grid: *rNM.TAG*.
6. Kinetic energy, magnetic energy and temperature/entropy **spectra**:
   - Kinetic energy: *kin_spec_#.TAG*,
   - Magnetic energy: *kin_spec_#.TAG*,
   - Velocity square: *u2_spec_#.TAG*,
   - Temperature/entropy: *T_spec_#.TAG*,
   - Time-averaged kinetic energy: *kin_spec_ave.TAG*,
   - Time-averaged magnetic energy: *mag_spec_ave.TAG*,
   - Time-averaged temperature/entropy: *T_spec_ave.TAG*,
   - 2-D ([r,ell] and [r,m]) spectra: *2D_[mag|kin|u2]_spec_#.TAG*.
7. Output snapshot that contains the 3-D components of the velocity field, the magnetic field and the temperature/entropy. Those files are named **graphic files** *G_#.TAG* (or *G_ave.TAG* for its time-averaged counterpart).
8. Time evolution of some chosen fields. Those files are named **movie files**: **_mov.TAG*.
9. Checkpoints outputs that will allow the code to restart. Those files are named **restart files**: *rst_end.TAG*.
10. **Time-evolution of the poloidal and toroidal coefficients** at diffent depths:
    - Time evolution of the poloidal magnetic field at the CMB: *B_coeff_cmb.TAG*,
    - Time evolution of the potentials at several depths: *[V|T|B]_coeff_r#.TAG*
11. **Additional specific outputs**:
    - Torsional oscillations (see *here*),
    - Potential files: *Vpot_#.TAG*, *Bpot_#.TAG* and *Tpot_#.TAG*,
    - Potential vorticity files: `PVZ.TAG` and `Vcy.TAG`,
    - Magnetic spectra for various radii: *rB[r|p]Spec.TAG*.

8.1 Log file: `log.TAG`

This is a text file contains information about the run, including many of the things which are printed to `STDOUT`. It has the following information in order of appearance:

- **Code version**: the version of the code
- **Parallelization**: information about number of MPI ranks being used, blocking information of OpenMP chunks and processor load balancing

- **Namelists**: displays values of all namelist variables. The ones input by the user should have the input values while the rest of them are set to their default values.
- **Mode** The mode of the run - self-consistent/kinematic dynamo, convection, couette flow etc. See the *control namelist* for more information about *mode*.
- **Grid parameters**: information about the grid sizes and truncation being used. More information about this in the *grid namelist*. If a new grid, different from that in the restart file is used, then a comparison is shown between old and new grid parameters and the user is informed that the data is being mapped from the old to the new grid.
- **Progress**: information about the progress of the run for every 10% of the run and the mean wall time for time step.
- **Writing of graphic, movie, restart and spectra files**: displays the time step and tells the user whenever a *G_#.TAG*, *rst_#.TAG* or *spectra* file or a *movie frame* is written disk.
- **Energies**: gives kinetic and magnetic energies (total, poloidal, toroidal, total density) at the end of the run.
- **Time averages**: this part gives time averaged kinetic and magnetic energies (total, poloidal, toroidal, total density) and time averaged parameters (Rm, Elsass, Rol etc.). If *l_average=.true.*, this section also provides information about average spectra and graphic files being written.
- **Wall times**: this is the last part of the log file and it provides information about the mean wall time for running different parts of the code. These values can be used to judge the speed and scaling capabilities of your computer.

Most of these informations can be parsed and stored into a python class using `MagicSetup`:

```
>>> # read log.N0m2
>>> stp = MagicSetup(nml='log.N0m2')
>>> print(stp.ek, stp.prmag) # print Ekman and magnetic Prandtl numbers
>>> print(stp.l_max) # print l_max
```

8.2 Default time-series outputs

8.2.1 `e_kin.TAG`

This file contains the kinetic energy of the outer core, defined by

$$
\begin{aligned}
E_k = \frac{1}{2}\int_V \tilde{\rho} u^2 \,\mathrm{d}V &= E_{pol} + E_{tor} \\
&= \frac{1}{2}\sum_{\ell,m}\ell(\ell+1)\int_{r_i}^{r_o}\frac{1}{\tilde{\rho}}\left[\frac{\ell(\ell+1)}{r^2}|W_{\ell m}|^2 + \left|\frac{\mathrm{d}W_{\ell m}}{\mathrm{d}r}\right|^2\right]\mathrm{d}r \\
&+ \frac{1}{2}\sum_{\ell,m}\ell(\ell+1)\int_{r_i}^{r_o}\frac{1}{\tilde{\rho}}|Z_{\ell m}|^2\,\mathrm{d}r
\end{aligned}
\tag{8.1}
$$

The detailed calculations are done in the subroutine `get_e_kin`. This file contains the following informations:

No. of column	Contents
1	time
2	poloidal energy
3	toroidal energy
4	axisymmetric poloidal energy
5	axisymmetric toroidal energy
6	equatorial symmetric poloidal energy
7	equatorial symmetric toroidal energy
8	equatorial symmetric and axisymmetric poloidal energy
9	equatorial symmetric and axisymmetric toroidal energy

This file can be read using *MagicTs* with the following options:

```
>>> # To stack all the e_kin.TAG files of the current directory
>>> ts = MagicTs(field='e_kin', all=True)
>>> # To only read e_kin.N0m2
>>> ts = MagicTs(field='e_kin', tag='N0m2')
```

8.2.2 e_mag_oc.TAG

This file contains the magnetic energy of the outer core, defined by

$$
\begin{aligned}
E_m = \frac{1}{2}\int_V B^2\,\mathrm{d}V = E_{pol} + E_{tor} \\
&= \frac{1}{2}\sum_{\ell,m}\ell(\ell+1)\int_{r_i}^{r_o}\left[\frac{\ell(\ell+1)}{r^2}|b_{\ell m}|^2 + \left|\frac{\mathrm{d}b_{\ell m}}{\mathrm{d}r}\right|^2\right]\mathrm{d}r \\
&+ \frac{1}{2}\sum_{\ell,m}\ell(\ell+1)\int_{r_i}^{r_o}|j_{\ell m}|^2\,\mathrm{d}r
\end{aligned}
\tag{8.2}
$$

The detailed calculations are done in the subroutine *get_e_mag*. This file contains the following informations:

No. of column	Contents
1	time
2	outer core poloidal energy
3	outer core toroidal energy
4	outer core axisymmetric poloidal energy
5	outer core axisymmetric toroidal energy
6	outside potential field energy
7	outside axisymmetric potential field energy
8	equatorial symmetric poloidal energy
9	equatorial symmetric toroidal energy
10	equatorial symmetric and axisymmetric poloidal energy
11	equatorial symmetric and axisymmetric toroidal energy
12	outside potential field energy
13	outside potential field axisymmetric energy

This file can be read using *MagicTs* with the following options:

```
>>> # To stack all the e_mag_oc.TAG files of the current directory
>>> ts = MagicTs(field='e_mag_oc', all=True)
>>> # To only read e_mag_oc.N0m2
>>> ts = MagicTs(field='e_mag_oc', tag='N0m2')
```

8.2.3 e_mag_ic.TAG

This file contains the magnetic energy of the inner core. The detailed calculations are done in the subroutine *get_e_mag*. This file contains the following informations:

No. of column	Contents
1	time
2	inner core poloidal energy
3	inner core toroidal energy
4	inner core axisymmetric poloidal energy
5	inner core axisymmetric toroidal energy

This file can be read using *MagicTs* with the following options:

```
>>> # To stack all the e_mag_ic.TAG files of the current directory
>>> ts = MagicTs(field='e_mag_ic', all=True)
>>> # To only read e_mag_ic.N0m2
>>> ts = MagicTs(field='e_mag_ic', tag='N0m2')
```

8.2.4 rot.TAG

This files contains the rotation of the inner core and the mantle. Output concerning the rotation of inner core and mantle. This file is written by the subroutine *write_rot*.

No. of column	Contents
1	time
2	Inner core rotation rate
3	Lorentz torque on inner core
4	viscous torque on inner core
5	mantle rotation rate
6	Lorentz torque on mantle
7	viscous torque on mantle

This file can be read using *MagicTs* with the following options:

```
>>> # To stack all the rot.TAG files of the current directory
>>> ts = MagicTs(field='rot', iplot=False, all=True)
```

8.2.5 dipole.TAG

This file contains several informations about the magnetic dipole. This file is written by the subroutine *get_e_mag*.

No. of column	Contents
1	time
2	tilt angle (colatitude in degrees) of the dipole
3	longitude (in degress) of dipole-pole
4	relative energy of the axisymmetric dipole
5	relative energy of the axisymmetric dipole at the CMB
6	energy of the axisymmetric dipole at the CMB normalized with the total energy up to spherical harmonic degree and order 11
7	relative energy of the total (axisymmetric and equatorial) dipole
8	relative energy of the total (axisymmetric and equatorial) dipole in the outer core
9	relative energy of the total dipole (axisymmetric and equatorial) at the CMB
10	energy of the total (axisymmetric and equatorial) dipole at the CMB
11	energy of the axisymmetric dipole at the CMB
12	energy of the dipole
13	energy of the axisymmetric dipole
14	magnetic energy at the CMB
15	magnetic energy up to spherical harmonic degree and order 11
16	ratio between equatorial dipole energy and equatorial poloidal energy
17	difference between energy at the CMB and equatorial symmetric energy at the CMB, normalized by energy at the CMB
18	difference between energy at the CMB and axisymmetric energy at the CMB, normalized by energy at the CMB
19	difference between total energy and equatorial symmetric part of the total energy, normalized by the total energy
20	difference between total energy and axisymmetric part of the total energy, normalized by the total energy

This file can be read using *MagicTs* with the following options:

```
>>> # To stack all the dipole.TAG files of the current directory
>>> ts = MagicTs(field='dipole', all=True)
```

8.2.6 par.TAG

This files contains the outputs of several parameters that describe flow and magnetic fields (Reynolds number, Elsasser number, flow lengthscales, etc.). This file is written by the subroutine *output*.

No. of column	Contents
1	time
2	(magnetic) Reynolds number
3	Elsasser number
4	Local Rossby number Rol
5	Realtive geostrophic kinetic energy
6	Total dipolarity
7	CMB dipolarity
8	Axial flow length scale dlV
9	Flow length scale dmV
10	Flow length scale dpV
11	Flow length scale dzV
12	Dissipation length scale lvDiss
13	Dissipation length scale lbDiss
14	Magnetic length scale dlB
15	Magnetic length scale dlB
16	Elsasser number at CMB
17	Local Rol based on non-ax. flow
18	Convective flow length scale dlVc
19	CMB zonal flow at the equator

This file can be read using *MagicTs* with the following options:

```
>>> # To stack all the par.TAG files of the current directory
>>> ts = MagicTs(field='par', all=True)
```

8.2.7 misc.TAG

This files contains informations about heat transfer (Nusselt number and temperature at both boundaries), as well as various additional informations (helicity, geostrophy, etc.). This file is written by the subroutine *outMisc*.

Note: The columns 6-9 and 17-20 are **only** calculated when *l_hel=.true.*. The columns 10-16 are only calculated when *l_par=.true.*.

No. of column	Contents
1	time
2	Nusselt number at the inner core
3	CMB Nusselt number at the CMB
4	Entropy at the inner core
5	Entropy at the CMB
6	Helicity (northern hemisphere)
7	Helicity (southern hemisphere)
8	RMS helicity (northern hemisphere)
9	RMS helicity (southern hemisphere)
10	Relative geostrophic kinetic energy
11	Relative kinetic energy in the northern part of the TC
12	Relative kinetic energy in the southern part of the TC
13	Kinetic energy
14	North/South correlation of Vz, outside the TC
15	North/South correlation of vorticity outside the TC
16	North/South correlation of helicity outside the TC

This file can be read using *MagicTs* with the following options:

```
>>> # To stack all the misc.TAG files of the current directory
>>> ts = MagicTs(field='misc', all=True)
```

8.3 Additional optional time-series outputs

8.3.1 AM.TAG

Note: This file is **only** written when *l_AM=.true.*

This file contains the time series of the angular momentum of the inner core, the outer core and the mantle. This file is written by the subroutine *write_rot*.

No. of column	Contents
1	time
2	angular momentum of the outer core
3	angular momentum of the inner core
4	angular momentum of the mantle
5	total angular momentum
6	relative in angular momentum, per time step
7	total kinetic angular momentum
8	relative change in kinetic energy, per time step
9	kinetic angular momentum of the inner core
10	kinetic angular momentum of the outer core
11	kinetic angular momentum of the mantle

This file can be read using *MagicTs* with the following options:

```
>>> # To stack all the AM.TAG files of the current directory
>>> ts = MagicTs(field='AM', all=True)
```

8.3.2 power.TAG

Note: This file is **only** written when *l_power=.true.*

This file contains the power budget diagnostic. This file is computed by the subroutine *get_power*.

No. of column	Contents
1	time
2	Buoyancy power: $Ra\, g(r)\, \langle u_r T' \rangle_s$
3	Viscous power at the inner boundary (ICB)
4	Viscous power at the outer boundary (CMB)
5	Viscous dissipation: $\langle (\nabla \times u)^2 \rangle_s$
6	Ohmic dissipation: $\langle (\nabla \times B)^2 \rangle_s$
7	Total power at the CMB (viscous + Lorentz)
8	Total power at the ICB (viscous + Lorentz)
9	Total power
10	Time variation of total power

This file can be read using *MagicTs* with the following options:

```
>>> # To stack the files that match the pattern  ``power.N0m2*``
>>> ts = MagicTs(field='power', tags='N0m2*')
```

8.3.3 dtE.TAG

Note: This file is **only** written when *l_power=.true.*

This file contains the time-derivatives of the total energy. It allows to accurately monitor how the total energy varies with time. This file is generated by the subroutine `output`.

No. of column	Contents
1	time
2	time-derivative of the total energy $\partial E/\partial t$
3	integrated time variation of the total energy
4	relative time variation of the total energy

8.3.4 u_square.TAG

Note: This file is **only** written in anelastic models, i.e. either when *strat/=0* or when *interior_model/="None"*

This file contains the square velocity of the outer core. It is actually very similar to the *e_kin.TAG* file, except that the density background $\tilde{\rho}$ is removed:

$$\begin{aligned}\mathcal{U} = \frac{1}{2}\int_V u^2\,\mathrm{d}V &= \mathcal{U}_{pol} + \mathcal{U}_{tor} \\ &= \frac{1}{2}\sum_{\ell,m}\ell(\ell+1)\int_{r_i}^{r_o}\frac{1}{\tilde{\rho}^2}\left[\frac{\ell(\ell+1)}{r^2}|W_{\ell m}|^2 + \left|\frac{\mathrm{d}W_{\ell m}}{\mathrm{d}r}\right|^2\right]\mathrm{d}r \\ &+ \frac{1}{2}\sum_{\ell,m}\ell(\ell+1)\int_{r_i}^{r_o}\frac{1}{\tilde{\rho}^2}|Z_{\ell m}|^2\,\mathrm{d}r\end{aligned}$$

The detailed calculations are done in the subroutine `get_u_square`. This file contains the following informations:

No. of columns	Contents
1	time
2	poloidal part $\mathcal{U}_{pol}$
3	toroidal part $\mathcal{U}_{pol}$
4	axisymmetric contribution to the poloidal part
5	axisymmetric contribution to the toroidal part
6	Rossby number: $Ro = E\sqrt{\frac{2\mathcal{U}}{V}}$
7	Magnetic Reynolds number: $Rm = Pm\sqrt{\frac{2\mathcal{U}}{V}}$
8	local Rossby number: $Ro_l = Ro\frac{d}{l}$
9	average flow length scale: l
10	local Rossby number based on the non-axisymmetric components of the flow
11	average flow length scale based on the non-axisymmetric components of the flow

This file can be read using `MagicTs` with the following options:

```
>>> # To stack all the u_square.TAG files of the current directory
>>> ts = MagicTs(field='u_square', all=True)
```

8.3.5 drift[V|B][D|Q].TAG

Note: These files are **only** written when *l_drift=.true.*

These files store spherical harmonic coefficients of the toroidal (poloidal) potential of the flow (magnetic) field, only for $\ell = m$ or $\ell = m + 1$ depending on the symmetry - `D` for **D** ipolar and `Q` for **Q** uadrupolar. The coefficients are stored at different three different radial levels - `n_r1, nr_2, n_r3` for the velocity and two different radial levels - `n_r1` and `n_r2` - for the magnetic field.

The symmetries can be summarized below:

Field	Dipolar	Quadrupolar
Velocity	$\ell = m$	$\ell = m + 1$
Magnetic	$\ell = m + 1$	$\ell = m$

$\ell+m =$ even for toroidal potential refers to an equatorially antisymmetric field (*Dipolar*), while the same for a poloidal potential is associated with an equatorially symmetric field (*Quadrupolar*). The sense is opposite when $\ell + m =$ odd. This is the reason for the choice of selecting these specific coefficients.

The columns of the files look like follows:

For the flow field:

- n_r1 = (1/3) * *n_r_max-1*
- n_r2 = (2/3) * *n_r_max-1*
- n_r3 = *n_r_max-1*

Column no.	DriftVD.TAG	DriftVQ.TAG
1	Time	Time
2	z (*minc*, minc) at n_r1	z (*minc+1*, minc) at n_r1
3	z (2*minc, 2*minc) at n_r1	z (2*minc+1, 2*minc) at n_r1
4	z (3*minc, 3*minc) at n_r1	z (3*minc+1, 3*minc) at n_r1
5	z (4*minc, 4*minc) at n_r1	z (4*minc+1, 4*minc) at n_r1
6	z (minc, minc) at n_r2	z (minc+1, minc) at n_r2
7	z (2*minc, 2*minc) at n_r2	z (2*minc+1, 2*minc) at n_r2
8	z (3*minc, 3*minc) at n_r2	z (3*minc+1, 3*minc) at n_r2
9	z (4*minc, 4*minc) at n_r2	z (4*minc+1, 4*minc) at n_r2
10	z (minc, minc) at n_r3	z (minc+1, minc) at n_r3
11	z (2*minc, 2*minc) at n_r3	z (2*minc+1, 2*minc) at n_r3
12	z (3*minc, 3*minc) at n_r3	z (3*minc+1, 3*minc) at n_r3
13	z (4*minc, 4*minc) at n_r3	z (4*minc+1, 4*minc) at n_r3

For the magnetic field:

- n_r1 = *n_r_ICB*
- n_r2 = *n_r_CMB*

Column no.	DriftBD.TAG	DriftBQ.TAG
1	Time	Time
2	b (*minc+1*, minc) at n_r1	b (*minc*, minc) at n_r1
3	b (2*minc+1, 2*minc) at n_r1	b (2*minc, 2*minc) at n_r1
4	b (3*minc+1, 3*minc) at n_r1	b (3*minc, 3*minc) at n_r1
5	b (4*minc+1, 4*minc) at n_r1	b (4*minc, 4*minc) at n_r1
6	b (minc+1, minc) at n_r2	b (minc, minc) at n_r2
7	b (2*minc+1, 2*minc) at n_r2	b (2*minc, 2*minc) at n_r2
8	b (3*minc+1, 3*minc) at n_r2	b (3*minc, 3*minc) at n_r2
9	b (4*minc+1, 4*minc) at n_r2	b (4*minc, 4*minc) at n_r2

Analysis of these files can give you information about the drift frequency of the solution and it's symmetry.

8.3.6 `iner[P|T].TAG`

Note: These files are **only** written when *l_iner=.true.* and *minc = 1*.

These files contain time series of spherical harmonic coefficients upto degree, $\ell = 6$ at a radius $r = (r_{cmb} - r_{icb})/2$. The `inerP.TAG` contains coefficients of the poloidal potential while the `inerT.TAG` contains coefficients of the toroidal potential.These files are written by the subroutine *write_rot*. The oscillations of these coefficients can be analysed to look for inertial modes. The columns of the `inerP.TAG` look like follows:

No. of column	Coefficient
1	$w(\ell = 1, m = 1)$
2	$w(\ell = 2, m = 1)$
3	$w(\ell = 2, m = 2)$
4	$w(\ell = 3, m = 1)$
...	
20	$w(\ell = 6, m = 5)$
21	$w(\ell = 6, m = 6)$

where $w(\ell, m)$ is the poloidal potential with degree ℓ and order m.

The columns of the `inerT.TAG` follow the following structure:

No. of column	Coefficient
1	$z(\ell = 1, m = 1)$
2	$z(\ell = 2, m = 1)$
3	$z(\ell = 2, m = 2)$
4	$z(\ell = 3, m = 1)$
...	
20	$z(\ell = 6, m = 5)$
21	$z(\ell = 6, m = 6)$

where $z(\ell, m)$ is the toroidal potential with degree ℓ and order m.

8.3.7 `SR[IC|MA].TAG`

Note: These files are **only** written for *nRotIc=-1* (for `SRIC.TAG`) or *nRotMa=-1* (for `SRMA.TAG`). In other words, these outputs are produced **only** when one of the boundaries is made to rotate at a prescribed rotation rate.

These files contain information about power due to torque from viscous and Lorentz forces at the inner core boundary (`SRIC.TAG`) or core mantle boundary (`SRMA.TAG`).The columns look like follows:

No. of column	Contents
1	Time
2	$\Omega_{IC}\|\Omega_{MA}$
3	Total power = Lorentz + Viscous
4	Viscous power
5	Lorentz force power

8.3.8 `dtVrms.TAG`

Note: This file is **only** written when *l_RMS=.true.*

This files contains the RMS force balance of the Navier Stokes equation. This file is written by the subroutine *dtVrms*.

No. of column	Contents
1	time
2	Flow changes: inertia–advection
3	Coriolis force
4	Lorentz force
5	Advection term
6	Diffusion term
7	Buoyancy term
8	Pressure gradient term
9	Sum of force terms: geostrophic balance
10	Sum of force terms: magnetostrophic balance
11	Sum of force terms: Archemidian balance
12	Sum of force terms: Lorentz/Coriolis
13	Sum of force terms: Pressure/Lorentz
14	Sum of force terms: Coriolis/Inertia/Archimedean

This file can be read using *MagicTs* with the following options:

```
>>> # To stack all the dtVrms.TAG files of the current directory
>>> ts = MagicTs(field='dtVrms', all=True)
```

8.3.9 `dtBrms.TAG`

Note: This file is **only** written when *l_RMS=.true.*

This files contains the RMS terms that enter the induction equation. This file is written by the subroutine *dtBrms*.

No. of column	Contents
1	time
2	Changes in magnetic field (poloidal)
3	Changes in magnetic field (toroidal)
4	Poloidal induction term
5	Toroidal induction term
8	Poloidal diffusion term
9	Toroidal diffusion term
10	Omega effect / toroidal induction term
11	Omega effect
12	Production of the dipole field
13	Production of the axisymmetric dipole field

This file can be read using *MagicTs* with the following options:

```
>>> # To stack all the dtBrms.TAG files of the current directory
>>> ts = MagicTs(field='dtBrms', all=True)
```

8.3.10 perpPar.TAG

Note: This file is **only** written when *l_perpPar=.true.*

This file contains several time series that decompose the kinetic energy into components parallel and perpendicular to the rotation axis. This file is calculated by the subroutine *outPerpPar*.

No. of column	Contents
1	radial level
2	Total kinetic energy perpendicular to the rotation axis: $\frac{1}{2}\langle u_s^2 + u_\phi^2 \rangle_V$
3	Total kinetic energy parallel to the rotation axis: $\frac{1}{2}\langle u_z^2 \rangle_V$
4	Axisymmetric kinetic energy perpendicular to the rotation axis
5	Axisymmetric kinetic energy parallel to the rotation axis

This file can be read using *MagicTs* with the following options:

```
>>> # To stack all the perpPar.TAG files of the current directory
>>> ts = MagicTs(field='perpPar', all=True)
```

8.4 Time-averaged radial profiles

8.4.1 eKinR.TAG

This file contains the time and horizontally averaged outer core kinetic energy along the radius. This file is calculated by the subroutine *get_e_kin*.

No. of column	Contents
1	radial level
2	time and horizontally averaged poloidal energy
3	time and horizontally averaged axisymmetric poloidal energy
4	time and horizontally averaged toroidal energy
5	time and horizontally averaged axisymmetric toroidal energy
6	time and horizontally averaged poloidal energy, normalized by surface area at this radial level
7	time and horizontally averaged axisymmetric poloidal energy, normalized by surface area at this radial level
8	time and horizontally averaged toroidal energy, normalized by surface area at this radial level
9	time and horizontally averaged axisymmetric toroidal energy, normalized by surface area at this radial level

This file can be read using *MagicRadial* with the following options:

```
>>> rad = MagicRadial(field='eKinR')
```

8.4.2 eMagR.TAG

This file contains the time and horizontally averaged outer core magnetic energy along the radius. This file is calculated by the subroutine *get_e_mag*.

No. of column	Contents
1	radial level
2	time and horizontally averaged poloidal energy
3	time and horizontally averaged axisymmetric poloidal energy
4	time and horizontally averaged toroidal energy
5	time and horizontally averaged axisymmetric toroidal energy
6	time and horizontally averaged poloidal energy, normalized by surface area at this radial level
7	time and horizontally averaged axisymmetric poloidal energy, normalized by surface area at this radial level
8	time and horizontally averaged toroidal energy, normalized by surface area at this radial level
9	time and horizontally averaged axisymmetric toroidal energy, normalized by surface area at this radial level
10	ratio between time-averaged dipole energy and time-averaged total energy

This file can be read using *MagicRadial* with the following options:

```
>>> rad = MagicRadial(field='eMagR')
```

8.4.3 parR.TAG

This file contains several time and horizontally averaged flow properties (magnetic Reynolds number, Rossby number, etc.). This file is calculated by the subroutine *outPar*.

No. of column	Contents
1	radial level
2	Magnetic Reynolds number
3	Local Rossby number (based on the mass-weighted velocity)
4	Local Rossby number (based on the RMS velocity)
5	Local flow length-scale (based on the mass-weighted velocity)
6	Local flow length-scale based on the non-axisymmetric flow components (based on the mass-weighted velocity)
7	Local flow length-scale (based on the RMS velocity)
8	Local flow length-scale based on the non-axisymmetric flow components (based on the RMS velocity)

This file can be read using *MagicRadial* with the following options:

```
>>> rad = MagicRadial(field='parR')
```

8.4.4 powerR.TAG

Note: This file is **only** written when *l_power=.true.*

This file contains the time and horizontally averaged power input (Buoyancy power) and outputs (viscous and Ohmic heating). This file is calculated by the subroutine *get_power*.

No. of column	Contents
1	radial level
2	Buoyancy power: $Ra\, g(r)\, \langle u_r T' \rangle_s$
3	Viscous dissipation: $\langle (\nabla \times u)^2 \rangle_s$
4	Ohmic dissipation: $\langle (\nabla \times B)^2 \rangle_s$

This file can be read using *MagicRadial* with the following options:

```
>>> rad = MagicRadial(field='powerR')
```

8.4.5 fluxesR.TAG

Note: This file is **only** written when *l_fluxProfs=.true.*

This file contains the time and horizontally averaged heat flux carried out by several physical processes: conductive flux, convective flux, kinetic flux, viscous flux, Poynting flux and resistive flux. This file is calculated by the subroutine *outPar*.

No. of column	Contents
1	radial level
2	**conductive flux:** $$\mathcal{F}_{cond} = -\frac{1}{Pr}\kappa\tilde{\rho}\tilde{T}\frac{\partial\langle s\rangle_s}{\partial r}$$
3	**convective flux:** $$\mathcal{F}_{conv} = \tilde{\rho}\tilde{T}\langle s\, u_r\rangle_s + \frac{Pr\, Di}{E\, Ra}\langle p\, u_r\rangle_s$$
4	**kinetic flux:** $$\mathcal{F}_{kin} = \frac{1}{2}\frac{Pr\, Di}{Ra}\langle u_r(\tilde{\rho}u^2)\rangle_s$$
5	**viscous flux:** $$\mathcal{F}_{visc} = -\frac{Pr\, Di}{Ra}\langle \boldsymbol{u}\cdot S\rangle_s$$
6	**Poynting flux:** $$\mathcal{F}_{poyn} = -\frac{Pr\, Di}{Ra\, E\, Pm}\langle(\boldsymbol{u}\times\boldsymbol{B})\times\boldsymbol{B}\rangle_s$$
7	**resistive flux:** $$\mathcal{F}_{poyn} = \frac{Pr\, Di}{Ra\, E\, Pm^2}\langle(\boldsymbol{\nabla}\times\boldsymbol{B})\times\boldsymbol{B}\rangle_s$$

This file can be read using *MagicRadial* with the following options:

```
>>> rad = MagicRadial(field='fluxesR')
```

8.4.6 bLayersR.TAG

Note: This file is **only** written when *l_viscBcCalc=.true.*

This file contains several time and horizontally averaged profiles that can be further used to determine thermal and viscous boundary layers: entropy (or temperature), entropy variance, horizontal velocity, radial derivative of the horizontal velocity, thermal dissipation rate. This file is calculated by the subroutine *outPar*.

No. of column	Contents
1	radial level
2	entropy: $\langle s \rangle_s$
3	**entropy variance:** $$\sqrt{\langle (s - \langle s \rangle_s)^2 \rangle_s}$$
4	**horizontal velocity:** $$u_h = \left\langle \sqrt{u_\theta^2 + u_\phi^2} \right\rangle_s$$
5	**radial derivative of the horizontal velocity:** $$\partial u_h / \partial r$$
6	**thermal dissipation rate:** $$\epsilon_T = \langle (\nabla T)^2 \rangle_s$$

This file can be read using *MagicRadial* with the following options:

```
>>> rad = MagicRadial(field='bLayersR')
```

Additional analyses of the boundary layers can then be carried out using *BLayers*:

```
>>> bl = BLayers(iplot=True)
```

8.4.7 perpParR.TAG

Note: This file is **only** written when *l_perpPar=.true.*

This file contains several time and horizontally averaged profiles that decompose the kinetic energy into components parallel and perpendicular to the rotation axis. This file is calculated by the subroutine *outPerpPar*.

No. of column	Contents
1	radial level
2	**Total kinetic energy perpendicular to the rotation axis:** $$\frac{1}{2}\langle u_s^2 + u_\phi^2 \rangle_s$$
3	**Total kinetic energy parallel to the rotation axis:** $$\frac{1}{2}\langle u_z^2 \rangle_s$$
4	Axisymmetric kinetic energy perpendicular to the rotation axis
5	Axisymmetric kinetic energy parallel to the rotation axis

This file can be read using *MagicRadial* with the following options:

```
>>> rad = MagicRadial(field='perpParR')
```

8.5 Transport properties of the reference state

These files define the radial transport properties of the reference state. These arrays are calculated in the subroutines *radial* and *transportProperties*. The output files are written in the subroutine *preCalc*.

8.5.1 anel.TAG

Note: This output is only calculated when an anelastic model is run, that is when *l_anel=.true.* or *l_anelastic_liquid=.true.*.

This file contains the radial profiles of the reference state (density, temperature, gravity, etc.).

No. of column	Contents
1	radial level: r
2	temperature: $\tilde{T}(r)$
3	density: $\tilde{\rho}(r)$
4	radial derivative of the density: $\beta = \mathrm{d}\ln\tilde{\rho}/\mathrm{d}r$
5	radial derivative of β: $\mathrm{d}\beta/\mathrm{d}r$
6	gravity: $g(r)$
7	entropy gradient: $\mathrm{d}s_0/\mathrm{d}r$
8	thermal diffusion operator: $\nabla\cdot(K(r)\tilde{T}(r)\nabla s_0)$

This file can be read using *MagicRadial* with the following options:

```
>>> rad = MagicRadial(field='anel')
>>> # print radius and density
>>> print(rad.radius, rad.rho0)
```

8.5.2 varCond.TAG

Note: This output is only calculated when the electrical conductivity varies with radius, i.e. when *nVarCond /= 0*

This file contains the radial profiles of the electrical conductivity, the electrical diffusivity and its radial derivative.

No. of column	Contents
1	radial level: r
2	electrical conductivity: $\sigma(r)$
3	electrical diffusivity: $\lambda(r) = 1/\sigma(r)$
4	radial derivative of the electrical diffusivity: $\mathrm{d}\ln\lambda/\mathrm{d}r$

This file can be read using *MagicRadial* with the following options:

```
>>> rad = MagicRadial(field='varCond')
>>> print(rad.conduc) # Electrical conductivity
```

8.5.3 varDiff.TAG

Note: This output is only calculated when the thermal diffusivity varies with radius, i.e. when *nVarDiff /= 0*

This file contains the radial profiles of the thermal conductivity, the thermal diffusivity and its radial derivative.

No. of column	Contents
1	radial level: r
2	thermal conductivity: $K(r)$
3	thermal diffusivity: $\kappa(r) = K(r)/\tilde{\rho}(r)$
4	radial derivative of the electrical diffusivity: $\mathrm{d}\ln\kappa/\mathrm{d}r$
5	Prandtl number: $Pr(r) = \nu(r)/\kappa(r)$

This file can be read using *MagicRadial* with the following options:

```
>>> rad = MagicRadial(field='varDiff')
>>> print(rad.kappa) # Thermal diffusivity
```

8.5.4 varVisc.TAG

Note: This output is only calculated when the kinematic viscosity varies with radius, i.e. when *nVarVisc /= 0*

This file contains the radial profiles of the dynamic viscosity, the kinematic viscosity and its radial derivative.

No. of column	Contents
1	radial level: r
2	dynamic viscosity: $\mu(r)$
3	kinetmatic viscosity: $\nu(r) = \mu(r)/\tilde{\rho}(r)$
4	radial derivative of the kinematic viscosity: $\mathrm{d}\ln\nu/\mathrm{d}r$
5	Prandtl number: $Pr(r) = \nu(r)/\kappa(r)$
6	magnetic Prandtl number $Pm(r) = \nu(r)/\lambda(r)$

This file can be read using *MagicRadial* with the following options:

```
>>> rad = MagicRadial(field='varVisc')
>>> # print kinematic viscosity and Ekman
>>> print(rad.kinVisc, rad.ekman)
```

8.6 Nonlinear mapping of the Chebyshev grid

8.6.1 rNM.TAG

Note: This file is only written when *l_newmap=.true.*.

This file contains the profile of the radial mapping and its derivatives:

No. of column	Contents
1	Grid point index
2	Radius of a grid point
3	First derivative of the mapping at a grid point
4	Second derivative of the mapping at a grid point
5	Third derivative of the mapping at a grid point

8.7 Spectra

8.7.1 kin_spec_#.TAG

This file contains the kinetic energy spectra. This file is written by the subroutine *spectrum*.

No. of column	Contents
1	degree / order
2	Poloidal kinetic energy versus degree
3	Poloidal kinetic energy versus order
4	Toroidal kinetic energy versus degree
5	Toroidal kinetic energy versus order

This file can be read using *MagicSpectrum* with the following options:

```
>>> sp = MagicSpectrum(field='ekin')
```

8.7.2 mag_spec_#.TAG

This file contains the magnetic energy spectra. This file is written by the subroutine *spectrum*.

No. of column	Contents
1	degree / order
2	Poloidal magnetic energy in the outer core versus degree
3	Poloidal magnetic energy in the outer core versus order
4	Toroidal magnetic energy in the outer core versus degree
5	Toroidal magnetic energy in the outer core versus order
6	Poloidal magnetic energy in the inner core versus degree
7	Poloidal magnetic energy in the inner core versus order
8	Toroidal magnetic energy in the inner core versus degree
9	Toroidal magnetic energy in the inner core versus order
10	Poloidal magnetic energy at the CMB versus degree
11	Poloidal magnetic energy at the CMB versus order
12	Poloidal magnetic energy at the CMB

This file can be read using *MagicSpectrum* with the following options:

```
>>> sp = MagicSpectrum(field='emag')
```

8.7.3 u2_spec_#.TAG

Note: This file is **only** written in anelastic models, i.e. either when *strat/=0* or when *interior_model/="None"*

This file contains the spectra of the square velocity. This file is written by the subroutine *spectrum*.

No. of column	Contents
1	degree / order
2	Poloidal contribution per degree in the outer core
3	Poloidal contribution per order in the outer core
4	Toroidal contribution per degree in the outer core
5	Toroidal contribution per order in the outer core

This file can be read using *MagicSpectrum* with the following options:

```
>>> # To read the file ``u2_spec_1.test``:
>>> sp = MagicSpectrum(field='u2', ispec=1, tag='test')
```

8.7.4 T_spec_#.TAG

This file contains the temperature/entropy spectra. It is written by the subroutine *spectrum_temp*.

No. of column	Contents
1	degree / order
2	RMS temperature/entropy versus degree
3	RMS temperature/entropy versus order
4	RMS temperature/entropy at the ICB versus degree
5	RMS temperature/entropy at the ICB versus order
6	RMS radial derivative of temperature/entropy at the ICB versus degree
7	RMS radial derivative of temperature/entropy at the ICB versus order

8.7.5 2D spectra *[2D_kin|mag|u2_spec]_#.TAG*

Those files contain 2-D spectra in the (r, ℓ) and in the (r, m) planes. In other words, the poloidal and toroidal energies versus degree ℓ or versus order m are computed for all radii. There are three kinds of those files that correspond to the aforementioned spectra, namely **2D_kin_spec_#.TAG**, **2D_mag_spec_#.TAG** and **2D_u2_spec_#.TAG**. The calculations are done in the subroutine *spectrum*. The structure of the output files are same for these three outputs. They are stored as fortran unformatted files.

Unformatted files are not directly human readable, and are used to store binary data and move it around without changing the internal representation. In fortran, the open, read and write operations for these files are performed as follows:

```
open(unit=4, file='test', form='unformatted')
read(unit=4) readVar
write(unit=n_out, iostat=ios) writeVar !Unformatted write
```

The structure of the 2D spectra files are as follows:

```
    !-------------
    ! Line 1
    !-------------

    time, n_r_max, l_max, minc ! Time, resolution, max(\ell), azimuthal symmetry

    !-------------
    ! Line 2
    !-------------

    r(1), r(2), r(3), ..., r(n_r_max)                    ! Radius

    !-------------
    ! Line 3
    !-------------

    e_p_l(l=1,r=1), e_p_l(l=1,r=2), ..., e_p_l(l=1,r=n_r_max),       ! Poloidal energy
    ...                                                              ! versus degree
    e_p_l(l=l_max,r=1), e_p_l(l=l_max,r=2), ..., e_p_l(l=l_max,r=n_r_max),

    !-------------
```

```
    ! Line 4
    !-------------

    e_p_m(m=0,r=1), e_p_l(m=0,r=2), ..., e_p_l(m=1,r=n_r_max),       ! Poloidal energy
    ...                                                             ! versus order
    e_p_l(m=l_max,r=1), e_p_l(m=l_max,r=2), ..., e_p_l(m=l_max,r=n_r_max),

    !-------------
    ! Line 3
    !-------------

    e_t_l(l=1,r=1), e_t_l(l=1,r=2), ..., e_t_l(l=1,r=n_r_max),       ! Toroidal energy
    ...                                                             ! versus degree
    e_t_l(l=l_max,r=1), e_t_l(l=l_max,r=2), ..., e_t_l(l=l_max,r=n_r_max),

    !-------------
    ! Line 4
    !-------------

    e_t_m(m=0,r=1), e_t_l(m=0,r=2), ..., e_t_l(m=1,r=n_r_max),       ! Toroidal energy
    ...                                                             ! versus order
    e_t_l(m=l_max,r=1), e_t_l(m=l_max,r=2), ..., e_t_l(m=l_max,r=n_r_max),
```

Those files can be read using the python class *MagicSpectrum2D* with the following options:

```
>>> # Read the file 2D_mag_spec_3.ext
>>> sp = MagicRSpec(tag='ext', field=e_mag', ispec=3)
>>> # Print e_pol_l and e_tor_m
>>> print(sp.e_pol_l, sp.e_tor_m)
```

8.7.6 kin_spec_ave.TAG

Note: This file is **only** written when *l_average=.true.*

This file contains the time-average kinetic energy spectra as well as squared quantities to allow a possible further reconstruction of the standard deviation. This file is written by the subroutine *spectrum_average*.

No. of column	Contents
1	degree / order
2	Time-averaged poloidal kinetic energy versus degree
3	Time-averaged poloidal kinetic energy versus order
4	Time-averaged toroidal kinetic energy versus degree
5	Time-averaged toroidal kinetic energy versus order
6	Time-averaged poloidal kinetic energy square versus degree
7	Time-averaged poloidal kinetic energy square versus order
8	Time-averaged toroidal kinetic energy square versus degree
9	Time-averaged toroidal kinetic energy square versus order

This file can be read using *MagicSpectrum* with the following options:

```
>>> # To read the file ``kin_spec_ave.test``:
>>> sp = MagicSpectrum(field='kin', ave=True, tag='test')
```

8.7.7 mag_spec_ave.TAG

Note: This file is **only** written when *l_average=.true.* and the run is magnetic

This file contains the time-average magnetic energy spectra. This file is written by the subroutine *spectrum_average*.

No. of column	Contents
1	degree / order
2	Time-averaged poloidal magnetic energy in the outer core versus degree
3	Time-averaged poloidal magnetic energy in the outer core versus order
4	Time-averaged toroidal magnetic energy in the outer core versus degree
5	Time-averaged toroidal magnetic energy in the outer core versus order
6	Time-averaged poloidal magnetic energy at the CMB versus degree
7	Time-averaged poloidal magnetic energy at the CMB versus order
8	Time-averaged poloidal magnetic energy in the outer core + its standard deviation versus degree
9	Time-averaged poloidal magnetic energy in the outer core - its standard deviation versus degree
10	Time-averaged poloidal magnetic energy in the outer core + its standard deviation versus order
11	Time-averaged poloidal magnetic energy in the outer core - its standard deviation versus order
12	Time-averaged toroidal magnetic energy in the outer core + its standard deviation versus degree
13	Time-averaged toroidal magnetic energy in the outer core - its standard deviation versus degree
14	Time-averaged toroidal magnetic energy in the outer core + its standard deviation versus order
15	Time-averaged toroidal magnetic energy in the outer core - its standard deviation versus order
16	Time-averaged poloidal magnetic energy at the CMB + its standard deviation versus order
17	Time-averaged poloidal magnetic energy at the CMB - its standard deviation versus order

This file can be read using *MagicSpectrum* with the following options:

```
>>> # To read the file ``mag_spec_ave.test``:
>>> sp = MagicSpectrum(field='mag', ave=True, tag='test')
```

8.7.8 T_spec_ave.TAG

Note: This file is **only** written when *l_average=.true.*

This file contains the time-averaged temperature/entropy spectra and their standard deviation. It is written by the subroutine *spectrum_temp_average*.

No. of column	Contents
1	Spherical harmonic degree
2	Time-averaged RMS temperature/entropy versus degree
3	Standard deviation of the temperature/entropy versus degree
4	Time-averaged RMS temperature/entropy at the ICB versus degree
5	Standard deviation of the temperature/entropy at the ICB versus degree
6	Time-averaged temperature/entropy gradient at the ICB versus degree
7	Standard deviation of the temperature/entropy gradient at the ICB versus degree

8.8 Graphic files `G_#.TAG` and `G_ave.TAG`

These are fortran unformatted files containing 3D data (in the form vector_array(phi, theta, r)) which can be used to visualize the solution. They are written after a fixed number of time steps as specified by the user in the *Output Control namelist* using the parameters listed in the section on *output of graphic files*. In case *l_average* is set to `.true.`, then an average graphic file, named `G_ave.TAG`, containing time averaged values of 3D data, is also written at the end of the simulation.

These files are written in chunks of latitude for one radial level at a time by the subroutine *graphOut* or by *graphOut_mpi* depending on whether `USE_MPI` is set to `Yes` or `No` in the Makefile. The structure of the file looks like below:

```
    !-------------
    ! Line 1
    !-------------

    version               !Graphout_version_9 (using MPI without pressure)
                          !Graphout_version_10 (using MPI, with pressure)
                          !Graphout_version_7 (without MPI, without pressure)
                          !Graphout_version_8 (without MPI, with pressure)
    !----------
    ! Line 2
    !----------

    runid

    !----------
    ! Line 3
    !----------

    time, n_r_max, n_theta_max, n_phi_tot,          !time = Time of writing
    n_r_ic_max-1, minc, nThetasBs,                  !(Simulation time),
    ra, ek, pr, prmag,                              !nThetasBs = no. of
    radratio, sigma_ratio                           !theta blocks

    !----------
    ! Line 4
    !----------

    theta(1:n_theta_max)

    !-----------------------------------------------------------------------

    !---------------------------------------
    !Graphout_version_9/Graphout_version_10
    !---------------------------------------
```

```
! These versions are written when the code uses MPI (USE_MPI=yes). Parallel
! chunks of fields are written for different radial levels. Chunks in theta
! are written in parallel using OpenMP

!---------------
! Data
!---------------

!-------------
! Block N
!-------------

!------------
! Line 4 + N
!------------


n_r-1, r(n_r)/r(1), n_theta_start, n_theta_stop   !Radial index, radius in terms
                                                  !of r_cmb, start and stop of
                                                  !the theta block


!---------------
! Line 4 + (N+1)
!---------------

sr(1:n_phi_tot, n_theta_start:n_theta_stop, n_r)  !Entropy

!---------------
! Line 4 + (N+2)
!---------------

vr(1:n_phi_tot, n_theta_start:n_theta_stop, n_r)  !Radial velocity

!---------------
! Line 4 + (N+3)
!---------------

vt(1:n_phi_tot, n_theta_start:n_theta_stop, n_r)  !Theta component of velocity

!---------------
! Line 4 + (N+4)
!---------------

vp(1:n_phi_tot, n_theta_start:n_theta_stop, n_r)  !Zonal (phi component) of
                                                  !velocity

if (l_PressGraph):                                !If pressure is stored

  !---------------
  ! Line 4 + (N+5)
  !---------------

  pr(1:n_phi_tot, n_theta_start:n_theta_stop, n_r)  !pressure


if (l_mag):                                        !For a magnetic run

  !-----------------
```

```
      ! Line 4 + (N+5/6)
      !-----------------

      br(1:n_phi_tot, n_theta_start:n_theta_stop, n_r)  !Radial magnetic field

      !-----------------
      ! Line 4 + (N+6/7)
      !-----------------

      bt(1:n_phi_tot, n_theta_start:n_theta_stop, n_r)  !Theta component of
                                                        !magnetic field

      !-----------------
      ! Line 4 + (N+7/8)
      !-----------------

      bp(1:n_phi_tot, n_theta_start:n_theta_stop, n_r)  !Zonal (phi component)
                                                        !of magnetic field


    !---------------------------------------
    !Graphout_version_7/Graphout_version_8
    !---------------------------------------

    !This version is written when the code does not use MPI (USE_MPI=no).
    !Chunks in theta are written in parallel with OpenMP.

    !---------------
    ! Data
    !---------------

    !-------------
    ! Block N
    !-------------

    !---------------
    ! Line 4 + (N+1)
    !---------------

    n_r-1, r(n_r)/r(1), n_theta_start, n_theta_stop

    !------------------------------------------------------------
    ! Each of the following data point is written in a new line
    !------------------------------------------------------------

    !-----------------
    ! Entropy
    !-----------------

    sr(1,n_theta_start,n_r)           !n_phi = 1, n_theta = n_theta_start, n_r
    sr(2,n_theta_start,n_r)           !n_phi = 2, n_theta = n_theta_start, n_r
    ...
    sr(n_phi_tot,n_theta_start,n_r)   !n_phi = n_phi_tot, n_theta = n_theta_start, n_r
    sr(1,n_theta_start+1,n_r)         !n_phi = 1, n_theta = n_theta_start+1, n_r
    ...
    sr(n_phi_tot,n_theta_start+1,n_r)
    ...
    sr(1,n_theta_stop,n_r)            !n_phi = 1, n_theta = n_theta_stop, n_r
```

```
    sr(2,n_theta_stop,n_r)             !n_phi = 2, n_theta = n_theta_stop, n_r
    ...
    sr(n_phi_tot,n_theta_stop,n_r)     !n_phi = n_phi_tot, n_theta = n_theta_stop, n_r

    !-----------------
    ! Radial velocity
    !-----------------

    vr(1,n_theta_start,n_r)            !n_phi = 1, n_theta = n_theta_start, n_r
    vr(2,n_theta_start,n_r)            !n_phi = 2, n_theta = n_theta_start, n_r
    ...
    vr(n_phi_tot,n_theta_start,n_r)    !n_phi = n_phi_tot, n_theta = n_theta_start, n_r
    vr(1,n_theta_start+1,n_r)          !n_phi = 1, n_theta = n_theta_start+1, n_r
    ...
    vr(n_phi_tot,n_theta_start+1,n_r)
    ...
    vr(1,n_theta_stop,n_r)             !n_phi = 1, n_theta = n_theta_stop, n_r
    vr(2,n_theta_stop,n_r)             !n_phi = 2, n_theta = n_theta_stop, n_r
    ...
    vr(n_phi_tot,n_theta_stop,n_r)     !n_phi = n_phi_tot, n_theta = n_theta_stop, n_r

    !-----------------------------
    ! Theta component of velocity
    !-----------------------------

    vt(1,n_theta_start,n_r)            !n_phi = 1, n_theta = n_theta_start, n_r
    vt(2,n_theta_start,n_r)            !n_phi = 2, n_theta = n_theta_start, n_r
    ...
    vt(n_phi_tot,n_theta_start,n_r)    !n_phi = n_phi_tot, n_theta = n_theta_start, n_r
    vt(1,n_theta_start+1,n_r)          !n_phi = 1, n_theta = n_theta_start+1, n_r
    ...
    vt(n_phi_tot,n_theta_start+1,n_r)
    ...
    vt(1,n_theta_stop,n_r)             !n_phi = 1, n_theta = n_theta_stop, n_r
    vt(2,n_theta_stop,n_r)             !n_phi = 2, n_theta = n_theta_stop, n_r
    ...
    vt(n_phi_tot,n_theta_stop,n_r)     !n_phi = n_phi_tot, n_theta = n_theta_stop, n_r


    !---------------------------------
    ! Zonal (phi component) of velocity
    !---------------------------------

    vp(1,n_theta_start,n_r)            !n_phi = 1, n_theta = n_theta_start, n_r
    vp(2,n_theta_start,n_r)            !n_phi = 2, n_theta = n_theta_start, n_r
    ...
    vp(n_phi_tot,n_theta_start,n_r)    !n_phi = n_phi_tot, n_theta = n_theta_start, n_r
    vp(1,n_theta_start+1,n_r)          !n_phi = 1, n_theta = n_theta_start+1, n_r
    ...
    vp(n_phi_tot,n_theta_start+1,n_r)
    ...
    vp(1,n_theta_stop,n_r)             !n_phi = 1, n_theta = n_theta_stop, n_r
    vp(2,n_theta_stop,n_r)             !n_phi = 2, n_theta = n_theta_stop, n_r
    ...
    vp(n_phi_tot,n_theta_stop,n_r)     !n_phi = n_phi_tot, n_theta = n_theta_stop, n_r


    if (l_PressGraph):                 !If pressure is stored
```

```
!-----------
! Pressure
!-----------

pr(1,n_theta_start,n_r)           !n_phi = 1, n_theta = n_theta_start, n_r
pr(2,n_theta_start,n_r)           !n_phi = 2, n_theta = n_theta_start, n_r
...
pr(n_phi_tot,n_theta_start,n_r)   !n_phi = n_phi_tot, n_theta = n_theta_start, n_r
pr(1,n_theta_start+1,n_r)         !n_phi = 1, n_theta = n_theta_start+1, n_r
...
pr(n_phi_tot,n_theta_start+1,n_r)
...
pr(1,n_theta_stop,n_r)            !n_phi = 1, n_theta = n_theta_stop, n_r
pr(2,n_theta_stop,n_r)            !n_phi = 2, n_theta = n_theta_stop, n_r
...
pr(n_phi_tot,n_theta_stop,n_r)    !n_phi = n_phi_tot, n_theta = n_theta_stop, n_r


if (l_mag):                       !Only if it is a magnetic case

!----------------------
! Radial magnetic field
!----------------------

br(1,n_theta_start,n_r)           !n_phi = 1, n_theta = n_theta_start, n_r
br(2,n_theta_start,n_r)           !n_phi = 2, n_theta = n_theta_start, n_r
...
br(n_phi_tot,n_theta_start,n_r)   !n_phi = n_phi_tot, n_theta = n_theta_start, n_r
br(1,n_theta_start+1,n_r)         !n_phi = 1, n_theta = n_theta_start+1, n_r
...
br(n_phi_tot,n_theta_start+1,n_r)
...
br(1,n_theta_stop,n_r)            !n_phi = 1, n_theta = n_theta_stop, n_r
br(2,n_theta_stop,n_r)            !n_phi = 2, n_theta = n_theta_stop, n_r
...
br(n_phi_tot,n_theta_stop,n_r)    !n_phi = n_phi_tot, n_theta = n_theta_stop, n_r


!----------------------------------
! Theta component of magnetic field
!----------------------------------

bt(1,n_theta_start,n_r)           !n_phi = 1, n_theta = n_theta_start, n_r
bt(2,n_theta_start,n_r)           !n_phi = 2, n_theta = n_theta_start, n_r
...
bt(n_phi_tot,n_theta_start,n_r)   !n_phi = n_phi_tot, n_theta = n_theta_start, n_r
bt(1,n_theta_start+1,n_r)         !n_phi = 1, n_theta = n_theta_start+1, n_r
...
bt(n_phi_tot,n_theta_start+1,n_r)
...
bt(1,n_theta_stop,n_r)            !n_phi = 1, n_theta = n_theta_stop, n_r
bt(2,n_theta_stop,n_r)            !n_phi = 2, n_theta = n_theta_stop, n_r
...
bt(n_phi_tot,n_theta_stop,n_r)    !n_phi = n_phi_tot, n_theta = n_theta_stop, n_r


!--------------------------------------
! Zonal (phi component) of magnetic field
```

```
    !---------------------------------------

    bp(1,n_theta_start,n_r)            !n_phi = 1, n_theta = n_theta_start, n_r
    bp(2,n_theta_start,n_r)            !n_phi = 2, n_theta = n_theta_start, n_r
    ...
    bp(n_phi_tot,n_theta_start,n_r)    !n_phi = n_phi_tot, n_theta = n_theta_start, n_r
    bp(1,n_theta_start+1,n_r)          !n_phi = 1, n_theta = n_theta_start+1, n_r
    ...
    bp(n_phi_tot,n_theta_start+1,n_r)
    ...
    bp(1,n_theta_stop,n_r)             !n_phi = 1, n_theta = n_theta_stop, n_r
    bp(2,n_theta_stop,n_r)             !n_phi = 2, n_theta = n_theta_stop, n_r
    ...
    bp(n_phi_tot,n_theta_stop,n_r)     !n_phi = n_phi_tot, n_theta = n_theta_stop, n_r

    !-----------------
    !Subsequent blocks
    !-----------------

    !Block N+1 in both cases have data at the same radial level but the next
    !theta chunk (n_theta_start + nThetaB, n_theta_stop + n_thetaB)

    !After data for all the theta blocks have been written for one radial
    !level, everything above is repeated for the next radial level
```

The graphic files can be read using the python class *MagicGraph*.

```
>>> G = MagicGraph(ivar = 1, tag='TAG')
```

They can be visualized using the *Surf* class:

```
>>> s = Surf(tag='TAG')
>>> # Surface map of radial velocity:
>>> s.surf(field = 'vr', r = 0.5, cmap = 'jet', levels = 50)
>>> s.slice(field = 'br', lon_0 = [0]) # Longitudinal Slice of radial magnetic field
>>> s.equat(field = 'entropy')         # Equatorial slice of entropy
```

8.9 Movie files `*_mov.TAG`

Note: These files are written **only** when *l_movie = .true.* or when a finite number of movie frames are asked for using the input parameters described in the *standard inputs section* of the *output control namelist*.

These are unformatted fortran files containing time evolution of fields on different surfaces - constant radius, colatitude or azimuth or on the full 3D grid. The fields can be of various types like radial magnetic field or velocity, entropy, helicity etc. The type of field and the type of surface can be specified using a string that begins with the field name, followed by the surface type (or 'full 3D', when a 3D movie is desired). One such example is as follows:

```
    l_movie = .true.,
    n_movie_frames = 1000,
    movie(1) = "B r r=0.5",
    movie(2) = "V all 3D",
    movie(3) = "Hel Eq"
```

The code does not interpret any whitespaces and is not case-sensitive so there's no difference between, say, `B r cmb` and `brcmb`. For further details and a list of keywords for different fields and surfaces, please refer to the *movie* in the *output control namelist*.

These files are written by the subroutine *write_movie_frame*.

The movie files are suitably named to reflect the type of field and surface. Their names begin with the keyword for the type of movie asked for, followed by the type of surface, followed by the word 'mov'. Thus, a generic movie name looks like:

```
Keyword_SurType_mov.TAG
```

E.g: if one asks for the radial component of magnetic field on surface of CMB, the movie would be named as `Br_CMB_mov.TAG`.

When asks multiple movies for same surface types but different surface levels, the surfaces are numbered with integers. Thus, for the following namelist input,

```
l_movie = .true.,
n_movie_frames = 1000,
movie(1) = "B r r=0.5",
movie(2) = "V p r=0.5",
movie(3) = "V r r=0.8",
```

one would get the following movie files as output:

```
Br_R=C1_mov.TAG
Vp_R=C1_mov.TAG
Vr_R=C2_mov.TAG
```

The structure of a generic movie file is as follows:

```
!-------
! Line 1
!-------

version                         !Movie version: 'JW_Movie_Version_2'

!-------
! Line 2
!-------

n_type, n_surface,              !Type of movie,
const, n_fields                 !Type of surface (r,theta,phi,CMB,Eq etc.)

!-------
! Line 3
!-------

n_movie_field_type(1:n_fields, n_movie) !Type of fields (velocity,
                                        !mag field, vorticity etc.)
!-------
! Line 4
!-------

runid

!-------
! Line 5
!-------

n_r_mov_tot, n_r_max,           !Total number of
n_theta_max, n_phi_max,         !radial grid points (including IC),
minc, ra, ek, pr, prmag,        !grid data, physical parameters
radratio, tScale
```

```
!-------
! Line 6
!-------

r_mov_tot(1:n_r_mov_tot)/r_cmb !All radii in terms of r_CMB

!-------
! Line 7
!-------

theta(1:n_theta_max)           !All theta points

!-------
! Line 8
!-------

phi(1:n_phi_max)               !All phi points

!--------------------------------------------------------------------

!---------
! Frame N
!---------

!-----------
! Line 8 + N
!-----------

n_frame, t_movie(N), omega_ic, omega_ma, dipLat, dipLon, dipStr, dipStrGeo

!---------------
! Line 8 + (N+1)
!---------------

frame_data(1:n_fields,n_start:n_stop)  !Desired field data on a
                                       !surface or 3D volume
                                       !n_start = start index of a field
                                       !n_stop  = last index of a field

!-----------
! Frame N+1
!-----------

!---------------
! Line 8 + (N+2)
!---------------

n_frame, t_movie(N+1), omega_ic, omega_ma, dipLat, dipLon, dipStr, dipStrGeo

!---------------
! Line 8 + (N+3)
!---------------

frame_data(1:n_fields,n_start:n_stop)  !Desired field data on a
                                       !surface or 3D volume
                                       !n_start = start index of a field
                                       !n_stop  = last index of a field
```

```
...

!-----------
! Frame N+M                          !M is the desired number of movie frames
!-----------

!---------------
! Line 8 + (N+M)
!---------------

n_frame, t_movie(N+M), omega_ic, omega_ma, dipLat, dipLon, dipStr, dipStrGeo

!---------------
! Line 8 + (N+M)
!---------------

frame_data(1:n_fields,n_start:n_stop)   !Desired field data on a
                                        !surface or 3D volume
                                        !n_start = start index of a field
                                        !n_stop  = last index of a field
```

The 2D movie files can be read and displayed using the python class *Movie* as follows:

```
>>> Movie()    #Lists out available movie files to choose from
>>> M = Movie(file = 'Vr_R=C1_mov.TAG')
```

The 3D movie files can be read using the python class *Movie3D*:

```
>>> M = Movie3D(file = 'V_3D_mov.TAG')
```

8.10 Restart files `rst_*.TAG`

Note: These frequency of writing these files are determined by the standard inputs mentioned in the section on *restart files* in the *output control namelist*. If nothing is specified then, by default one restart file is written at the end of the run.

Note: A restart file is read **only** when *l_start = .true.*

These are unformatted fortran files containing a snapshot of information about spectral coefficients and physical and grid parameters. As the name suggests, these files are used to 'restart' a run from a specific time. One such file is read by the code at the beginning and are used as initial conditions for the run. These are very useful for continuing a simulation for a long time on computing clusters where the time for a single run is limited.

The file to be read at the beginning is specified by the input parameter *start_file* which takes in a string providing path to the file.

These files are written by the subroutine *store*.

The following notations will be used for the coefficients of potentials (note that scalar fields like temperature and pressure do not have a poloidal/toroidal decomposition):

Field	Poloidal	Toroidal
Magnetic	*b*	*aj*
Velocity	*w*	*z*
Temperature	*s*	
Pressure	*p*	

Time derivatives are denoted with a self-explanatory notation. e.g, `dbdt` is the first derivative of b.

The word `Last` appended to a variable name denotes that the value is of the time-step previous to the one during which the file is being written. They are needed for the time-stepping schemes.

`_ic` with a variable name says that it belongs to the Inner Core.

```
    !--------
    ! Line 1
    !--------

    time*tScale, dt*tScale, ra, pr, prmag, ek, radratio, inform, n_r_max,
    n_theta_max, n_phi_tot, minc, nalias, n_r_ic_max, sigma_ratio

    if (l_heat):                    !Run involving heat transport
                                    !(Convection)
    !---------
    ! Line 2
    !---------

       w,z,p,s

    !---------
    ! Line 3
    !---------


       dsdtLast,dwdtLast,dzdtLast,dpdtLast

    else:

    !---------
    ! Line 2
    !---------


       w,z,p

    !---------
    ! Line 3
    !---------

      dwdtLast,dzdtLast,dpdtLast

    if (l_mag):                                     !If magnetic run

    !---------
    ! Line 4
    !---------


      b, aj, dbdtLast, djdtLast

     if(l_mag .and. l_cond_ic):                     !If magnetic run
                                                    !and conducting inner core
    !---------
    ! Line 5
    !---------

      b_ic, aj_ic, dbdt_icLast, djdt_icLast
```

```
!-------------------------------------------------
! Line 4 or 5 or 6 depending on l_mag and l_cond_ic
!-------------------------------------------------

lorentz_torque_icLast, lorentz_torque_maLast, !Information about torques,
omega_ic1, omegaOsz_ic1, tOmega_ic1,          !prescribed rotation and
omega_ic2, omegaOsz_ic2, tOmega_ic2,          !oscillation rates,
omega_ma1, omegaOsz_ma1, tOmega_ma1,          !and the time step-size
omega_ma2, omegaOsz_ma2, tOmega_ma2,
dtNew
```

8.11 Poloidal and toroidal potentials at given depths

These are fortran unformatted files which store time series of poloidal and toroidal coefficients of different fields (magnetic field, velocity and temeperature) at specific depths.

In the following, `time(j)` is the time during the j^{th} time step, `time(N)` being the last step. `real` and `imag` denote real and imaginary parts, respectively, of spherical harmonic coefficients. Also, the following notations will be used for the coefficients of potentials (note that scalar fields like temperature do not have a poloidal/toroidal decomposition):

Field	Poloidal	Toroidal
Magnetic	*b*	*aj*
Velocity	*w*	*z*
Temperature	*s*	

First and second derivatives are denoted with a differential notation. e.g: *dw* is the first derivative of *w*, while *ddb* is the second derivative of *b*.

8.11.1 B_coeff_cmb.TAG

Note: This file is **only** written when *l_cmb_field=.true.*

This file contains time series of spherical harmonic coefficients for the poloidal potential of the magnetic field at the outer boundary (CMB) up to a spherical harmonic degree given by *l_max_cmb*. The detailed calculations are done in the subroutine *write_Bcmb*. The contents of the file look as follows:

- **Header** The file header consists of the information: *l_max_cmb*, *minc* and the number of data points `n_data`.
- **Data** Each chunk of data after the header has the same pattern of `time` followed by a list of real and imaginary values of coefficients.

Thus, on a whole, the structure of the file looks like follows:

```
!------------
! Line 1
!------------

l_max_cmb, minc, n_data

!------------------------------
...

!------------
! Line j + 1
!------------
```

```
    time(j),
    real(b(l=1,m=0)), imag(b(l=1,m=0)),
    real(b(l=2,m=0)), imag(b(l=2,m=0)),
    ...
    real(b(l=l_max_cmb,m=l_max_cmb)), imag(b(l=l_max_cmb,m=l_max_cmb)),

    ...

    !-------------
    ! Line N + 1
    !-------------

    time(N),
    real(b(l=1,m=0)), imag(b(l=1,m=0)),
    real(b(l=2,m=0)), imag(b(l=2,m=0)),
    ...
    real(b(l=l_max_cmb,m=l_max_cmb)), imag(b(l=l_max_cmb,m=l_max_cmb))
```

This file can be read using *MagicCoeffCmb* with the following options:

```
>>> # To stack the files B_cmb_coeff.testc to B_cmb_coeff.testf
>>> cmb = MagicCoeffCmb(tag='test[c-f]')
>>> # print Gauss coefficient for (\ell=10, m=3)
>>> print(cmb.glm[:, 10, 3])
```

8.11.2 Coefficients at desired radii

The following files **[B|V|T]_coeff_r#.TAG** save coefficients at specified depths and are written by the subroutine *write_coeff_r*. See the section on *CMB and radial coefficients* in the *ouput control namelist* for details of specifying depth, using *n_r_step* or *n_r_array* and desired maximum degree of output *l_max_r*. A separate file for each desired radius is written, numbered suitably as `[B|V|T]_coeff_r1.TAG`, `[B|V|T]_coeff_r2.TAG` etc.

8.11.3 B_coeff_r#.TAG

Note: This file is **only** written when *l_r_field=.true.*.

This file contains output of time series of the spherical harmonic coefficients of the poloidal and toroidal magnetic field potentials and the first and second derivatives of the poloidal potential coefficients in the order *b*, *db*, *aj* and *ddb*. The output is for a specific radius, r up to degree *l_max_r*.

- **Header** The file header consists of the information: *l_max_r*, *minc*, the number of data points `n_data` and the radius, *r*.
- **Data** Each chunk of data after the header contains the `time` at which the coefficients are stored, followed by the real and imaginary parts of: the poloidal coefficient `b`, it's first derivative *db*, the toroidal coefficient *aj* and the second derivative of the poloidal coefficient *ddb*.

The complete structure of the file looks like follows:

```
    !-------------
    ! Line 1
    !-------------

    l_max_r, minc, n_data, r
```

```
    !-------------------------------------------
    ...

    !------------
    ! Line j + 1
    !------------

    time(j),
    real(b(l=1,m=0)), imag(b(l=1,m=0)),
    real(b(l=2,m=0)), imag(b(l=2,m=0)),
    ...
    real(b(l=l_max_cmb,m=l_max_cmb)), imag(b(l=l_max_cmb,m=l_max_cmb)),
    real(db(l=1,m=0)), imag(db(l=1,m=0)),
    real(db(l=2,m=0)), imag(db(l=2,m=0)),
    ...
    real(db(l=l_max_cmb,m=l_max_cmb)), imag(db(l=l_max_cmb,m=l_max_cmb)),
    real(aj(l=1,m=0)), imag(aj(l=1,m=0)),
    real(aj(l=2,m=0)), imag(aj(l=2,m=0)),
    ...
    real(aj(l=l_max_cmb,m=l_max_cmb)), imag(aj(l=l_max_cmb,m=l_max_cmb)),
    real(ddb(l=1,m=0)), imag(ddb(l=1,m=0)),
    real(ddb(l=1,m=0)), imag(ddb(l=1,m=0)),
    ...
    real(ddb(l=l_max_cmb,m=l_max_cmb)), imag(ddb(l=l_max_cmb,m=l_max_cmb)),

    ...

    !------------
    ! Line N + 1
    !------------

    time(N),
    real(b(l=1,m=0)), imag(b(l=1,m=0)),
    real(b(l=2,m=0)), imag(b(l=2,m=0)),
    ...
    real(b(l=l_max_cmb,m=l_max_cmb)), imag(b(l=l_max_cmb,m=l_max_cmb)),
    real(db(l=1,m=0)), imag(db(l=1,m=0)),
    real(db(l=2,m=0)), imag(db(l=2,m=0)),
    ...
    real(db(l=l_max_cmb,m=l_max_cmb)), imag(db(l=l_max_cmb,m=l_max_cmb)),
    real(aj(l=1,m=0)), imag(aj(l=1,m=0)),
    real(aj(l=2,m=0)), imag(aj(l=2,m=0)),
    ...
    real(aj(l=l_max_cmb,m=l_max_cmb)), imag(aj(l=l_max_cmb,m=l_max_cmb)),
    real(ddb(l=0,m=0)), imag(ddb(l=0,m=0)),
    real(ddb(l=1,m=0)), imag(ddb(l=1,m=0)),
    ...
    real(ddb(l=l_max_cmb,m=l_max_cmb)), imag(ddb(l=l_max_cmb,m=l_max_cmb))
```

This file can be read using *MagicCoeffR* with the following options:

```
>>> # To stack the files B_coeff_r3.test* from the working directory
>>> cr = MagicCoeffR(tag='test*', field='B', r=3)
>>> # print the time and the poloidal potential for (\ell=3, m=3)
>>> print(cr.time, cr.wlm[:, 3, 3])
```

8.11.4 V_coeff_r#.TAG

Note: This file is **only** written when *l_r_field=.true.*

This file contains output of time series of the spherical harmonic coefficients of the poloidal and toroidal velocity field potentials and the first derivatives of the poloidal potential coefficients in the order *w*, *dw*, and *z*. The output is for a specific radius, r up to degree *l_max_r*.

- **Header** The file header consists of the information: *l_max_r*, *minc*, the number of data points `n_data` and the radius, *r*.
- **Data** Each chunk of data after the header contains the `time` at which the coefficients are stored, followed by the real and imaginary parts of: the poloidal coefficient *w*, it's first derivative *dw* and the toroidal coefficient *z*.

The complete structure of the file looks like follows:

```
!------------
! Line 1
!------------

l_max_r, minc, n_data, r

!-----------------------------------
...

!------------
! Line j + 1
!------------

time(j),
real(w(l=1,m=0)), imag(w(l=1,m=0)),
real(w(l=2,m=0)), imag(w(l=2,m=0)),
...
real(w(l=l_max_cmb,m=l_max_cmb)), imag(w(l=l_max_cmb,m=l_max_cmb)),
real(dw(l=1,m=0)), imag(dw(l=1,m=0)),
real(dw(l=2,m=0)), imag(dw(l=2,m=0)),
...
real(dw(l=l_max_cmb,m=l_max_cmb)), imag(dw(l=l_max_cmb,m=l_max_cmb)),
real(z(l=1,m=0)), imag(z(l=1,m=0)),
real(z(l=2,m=0)), imag(z(l=2,m=0)),
...
real(z(l=l_max_cmb,m=l_max_cmb)), imag(z(l=l_max_cmb,m=l_max_cmb)),

...

!--------------
! Line N + 1
!--------------

time(N),
real(w(l=1,m=0)), imag(w(l=1,m=0)),
real(w(l=2,m=0)), imag(w(l=2,m=0)),
...
real(w(l=l_max_cmb,m=l_max_cmb)), imag(w(l=l_max_cmb,m=l_max_cmb)),
real(dw(l=1,m=0)), imag(dw(l=1,m=0)),
real(dw(l=2,m=0)), imag(dw(l=2,m=0)),
...
real(dw(l=l_max_cmb,m=l_max_cmb)), imag(dw(l=l_max_cmb,m=l_max_cmb)),
real(z(l=1,m=0)), imag(z(l=1,m=0)),
```

```
    real(z(l=2,m=0)), imag(z(l=2,m=0)),
    ...
    real(z(l=l_max_cmb,m=l_max_cmb)), imag(z(l=l_max_cmb,m=l_max_cmb))
```

This file can be read using *MagicCoeffR* with the following options:

```
>>> # To stack the files V_coeff_r3.test* from the working directory
>>> cr = MagicCoeffR(tag='test*', field='V', r=3)
>>> # print the poloidal and toroidal potentials for (\ell=6, m=0)
>>> print(cr.wlm[:, 6, 0], cr.zlm[:, 6, 0])
```

8.11.5 T_coeff_r#.TAG

Note: This file is **only** written when *l_r_fieldT=.true.*

This file contains output of time series of the spherical harmonic coefficients of the temperature (or entropy) field. The output is for a specific radius, r up to degree *l_max_r*.

- **Header** The file header consists of the information: *l_max_r*, *minc*, the number of data points `n_data` and the radius, *r*.
- **Data** Each chunk of data after the header contains the `time` at which the coefficients are stored, followed by the real and imaginary parts of the coefficient *s*.

The complete structure of the file looks like follows:

```
    !------------
    ! Line 1
    !------------

    l_max_r, minc, n_data, r

    !----------------------------------

    ...

    !------------
    ! Line j + 1
    !------------

    time(j),
    real(s(l=0,m=0)), imag(s(l=0,m=0)),
    real(s(l=1,m=0)), imag(s(l=1,m=0)),
    real(s(l=2,m=0)), imag(s(l=2,m=0)),
    ...
    real(s(l=l_max_cmb,m=l_max_cmb)), imag(s(l=l_max_cmb,m=l_max_cmb)),

    !------------
    ! Line N + 1
    !------------

    time(N),
    real(s(l=0,m=0)), imag(s(l=0,m=0)),
    real(s(l=1,m=0)), imag(s(l=1,m=0)),
    real(s(l=2,m=0)), imag(s(l=2,m=0)),
    ...
    real(s(l=l_max_cmb,m=l_max_cmb)), imag(s(l=l_max_cmb,m=l_max_cmb)),
```

8.12 TO outputs

Note: These output files are **only** written when *l_TO=.true.*

8.12.1 TOZ_#.TAG and TOZM.TAG

8.12.2 TO_mov.TAG files

Note: This file is **only** written when *l_TOmovie=.true.*

This file contains the time evolution of the different forces that enter the phi-average of the azimuthal component of the Navier-Stokes equation. This is a special kind of *movie file* that contains seven different azimuthally-averaged fields in a (r, θ) plane : the axisymmetric zonal flow component , the azimuthal component of the Reynolds stresses, the azimuthal component of advection, the azimuthal component of viscosity, the azimuthal component of Lorentz force, the azimuthal component of Coriolis force and the azimuthal component of the time-derivative. The structure of the file is similar to a *movie file*, i.e. an unformatted fortran binary file with a header that describes the type of the movie file. The detailed calculations can be found in the subroutine *outTO*.

On a whole, the structure of the file looks like follows:

```
!----------
! Line 1
!----------

version

!----------
! Line 2
!----------

n_type, n_surface, const, n_fields

!----------
! Line 3
!----------

runid

!----------
! Line 4
!----------

n_r_movie_max, n_r_max, n_theta_max, n_phi_tot, minc, ra, ek, pr,
prmag, radratio, tScale

!----------
! Line 5
!----------

r(1), r(2), ..., r(n_r_movie_max)

!----------
! Line 6
!----------
```

```
    theta(1), theta(2), ..., theta(n_theta_max)

    !----------
    ! Line 7
    !----------

    phi(1), phi(2), ..., phi(n_theta_max)

    ...

    !----------
    ! Line 7+N
    !----------

    n_frame, t_movie(N), omega_ic, omega_ma, dipLat, dipLon, dipStr, dipStrGeo

    !----------
    ! Line 7+(N+1)
    !----------

    vphi(t=t_movie(N),phi=1,theta=1),
    vphi(t=t_movie(N),phi=1,theta=2),
    ...,
    vphi(t=t_movie(N),phi=n_phi_max,theta=n_theta_max)

    !----------
    ! Line 7+(N+2)
    !----------

    rey(t=t_movie(N),phi=1,theta=1),
    rey(t=t_movie(N),phi=1,theta=2),
    ...,
    rey(t=t_movie(N),phi=n_phi_max,theta=n_theta_max)

    ...

    !----------
    ! Line 7+(N+7)
    !----------

    dtVphi(t=t_movie(N),phi=1,theta=1),
    dtVphi(t=t_movie(N),phi=1,theta=2),
    ...,
    dtVphi(t=t_movie(N),phi=n_phi_max,theta=n_theta_max)
```

This file can be read using *TOMovie* with the following options:

```
>>> # To load 'TO_mov.test' and time-average it:
>>> to = TOMOvie(file='TO_mov.test', avg=True, levels=65, cm='seismic')
```

8.13 Radial spectra `rB[r|p]Spec.TAG`

Note: This files are **only** written when *l_rMagSpec=.true.*

Those files contain the time-evolution of the poloidal (**rBrSpec.TAG**) and the toroidal (**rBpSpec.TAG**) magnetic

energies for all radii including the inner core and for spherical harmonic degrees from $\ell = 1$ to $\ell = 6$. The calculations are done in the subroutines *rBrSpec* and *rBpSpec*, respectively. The outputs are stored as a fortran unformatted file which follows the following structure for `rBrSpec.TAG`:

```
!-------------
! Line N
!-------------

time[N],
(real(e_p(l=1,n_r),kind=outp),n_r=1,n_r_tot-1), ! Poloidal energy for \ell=0
(real(e_p(l=2,n_r),kind=outp),n_r=1,n_r_tot-1),
...
(real(e_p(l=6,n_r),kind=outp),n_r=1,n_r_tot-1)  ! Poloidal energy for \ell=6

!-------------
! Line N+1
!-------------

time[N],
(real(e_p_ax(l=1,n_r),kind=outp),n_r=1,n_r_tot-1), ! Pol. energy for \ell=0, m=0
(real(e_p_ax(l=2,n_r),kind=outp),n_r=1,n_r_tot-1),
...
(real(e_p_ax(l=6,n_r),kind=outp),n_r=1,n_r_tot-1)

!-------------
! Line N+2
!-------------

time[N+1]
...
```

The `rBpSpec.TAG` files have exactly the same structure (just replacing the poloidal energy by its toroidal counterpart).

Warning: Be careful that in this file, *n_r_tot* is the **total** number of grid points (thus including the inner core).

Those files can be read using the python class *MagicRSpec* with the following options:

```
>>> # Read the files BrSpec.testa, BrSpec.testb and BrSpec.testc and stack them
>>> rsp = MagicRSpec(tag='test[a-c]', field='Br')
>>> # Print time and the time evolution of e_pol(\ell=4) at the 10th radial grid point
>>> print(rsp.time, rsp.e_pol[:, 10, 3])
```

8.14 Potential files `[V|B|T]pot_#.TAG`

8.14.1 `Vpot_#.TAG` and `Vpot_ave.TAG`

Note: These output files are **only** written when either when *l_storePot=.true.* or when *l_storeVpot=.true.*.

These files contain a snapshot of the poloidal and toroidal flow potentials *w* and *z* in the Chebyshev space for all spherical harmonic degrees and orders. They basically contain two arrays of dimension (*lm_max*, *n_cheb_max*). The detailed calculations are done in the subroutine *storePot*. The outputs are stored as a fortran unformatted file which follows the following structure:

```
!-------------
! Line 1
!-------------

l_max, n_cheb_max, n_cheb_ic_max, minc, lm_max     ! Header (truncation)

!-------------
! Line 2
!-------------

ra, ek, pr, prmag, sigma_ratio, omega_ma, omega_ic ! Header (physics)

!------------
! Line 3
!-------------

time,                                              ! Time and poloidal potential
w(lm=1,n_cheb=1), w(lm=2, n_cheb=1), ..., w(lm=lm_max, n_cheb=1),
...
w(lm=1,n_cheb=n_cheb_max, ..., w(lm=lm_max,n_cheb=n_cheb_max)

!------------
! Line 4
!-------------

time,                                              ! Time and toroidal potential
z(lm=1,n_cheb=1), z(lm=2, n_cheb=1), ..., z(lm=lm_max, n_cheb=1),
...
z(lm=1,n_cheb=n_cheb_max, ..., z(lm=lm_max,n_cheb=n_cheb_max)
```

8.14.2 `Bpot_#.TAG`, `Bpot_ave.TAG`

Note: These output files are **only** written when either when *l_storePot=.true.* or when *l_storeBpot=.true.*.

These files contain a snapshot of the poloidal and toroidal magnetic potentials *b* and *aj* in the Chebyshev space for all spherical harmonic degrees and orders. The detailed calculations are done in the subroutine *storePot*. The outputs are stored as a fortran unformatted file which follows the following structure:

```
!-------------
! Line 1
!-------------

l_max, n_cheb_max, n_cheb_ic_max, minc, lm_max     ! Header (truncation)

!-------------
! Line 2
!-------------

ra, ek, pr, prmag, sigma_ratio, omega_ma, omega_ic ! Header (physics)

!------------
! Line 3
!-------------

time,                                              ! Time and poloidal potential
b(lm=1,n_cheb=1), b(lm=2, n_cheb=1), ..., b(lm=lm_max, n_cheb=1),
```

```
...
b(lm=1,n_cheb=n_cheb_max, ..., b(lm=lm_max,n_cheb=n_cheb_max)

!------------
! Line 4
!-------------

time,                                              ! Time and toroidal potential
aj(lm=1,n_cheb=1), aj(lm=2, n_cheb=1), ..., aj(lm=lm_max, n_cheb=1),
...
aj(lm=1,n_cheb=n_cheb_max, ..., aj(lm=lm_max,n_cheb=n_cheb_max)

!*********************************************************************!
! The two following lines are optional and are only written when there is  !
! an electrically-conducting inner-core                                    !
!*********************************************************************!

!------------
! Line 5
!-------------

time,                                              ! Time and poloidal potential
b_ic(lm=1,n_cheb=1), b_ic(lm=2, n_cheb=1), ..., b_ic(lm=lm_max, n_cheb=1),
...
b_ic(lm=1,n_cheb=n_cheb_max, ..., b_ic(lm=lm_max,n_cheb=n_cheb_max)

!------------
! Line 6
!-------------

time,                                              ! Time and toroidal potential
aj_ic(lm=1,n_cheb=1), aj_ic(lm=2, n_cheb=1), ..., aj_ic(lm=lm_max, n_cheb=1),
...
aj_ic(lm=1,n_cheb=n_cheb_max, ..., aj_ic(lm=lm_max,n_cheb=n_cheb_max)
```

8.14.3 `Tpot_#.TAG`, `Tpot_ave.TAG`

Note: These output files are **only** written when either when *l_storePot=.true.* or when *l_storeTpot=.true.*.

These files contain a snapshot of the temperature/entropy *s* in the spectral and Chebyshev spaces for all spherical harmonic degrees and orders. They basically contain one array of dimension (*lm_max*, *n_cheb_max*). The detailed calculations are done in the subroutine *storePot*. The outputs are stored as a fortran unformatted file which follows the following structure:

```
!-------------
! Line 1
!-------------

l_max, n_cheb_max, n_cheb_ic_max, minc, lm_max     ! Header (truncation)

!-------------
! Line 2
!-------------

ra, ek, pr, prmag, sigma_ratio, omega_ma, omega_ic ! Header (physics)
```

```
!------------
! Line 3
!-------------

time,                                                  ! Time and temperature/entropy
s(lm=1,n_cheb=1), s(lm=2, n_cheb=1), ..., s(lm=lm_max, n_cheb=1),
...
s(lm=1,n_cheb=n_cheb_max, ..., s(lm=lm_max,n_cheb=n_cheb_max)
```

CHAPTER

NINE

DATA VISUALISATION AND POST-PROCESSING

Most of the *output files* written during a run of MagIC can be treated with the python post-processing classes and functions present in the `$MAGIC_HOME/python/magic` directory. These classes depend on several python libraries that can be usually found in most of the Linux distributions.

9.1 Requirements

9.1.1 Hard dependencies

- python 2.7/3.3 or higher.
- matplotlib 1.0 or higher.
- scipy 0.10 or higher.

9.1.2 Optional dependencies

- Although entirely optional, the installation of ipython makes the interactive use of the post-processing python functions much more pleasant. Installing it is therefore recommanded for a smoother interactive usage of the python functions.
- The installation of the basemap toolkit is optional. If installed, additional projections for the *`magic.Surf`* (Aitoff, orthographic, Mollweide, etc.) class will be provided for 2-D surface plotting. Otherwise, the usage of *`magic.Surf`* is limited to the Hammer projection.

9.2 Configuration: `magic.cfg` file

A file named **magic.cfg** located in `$MAGIC_HOME/python/magic/magic.cfg` should have been created when you used the `source path/sourceme.sh` command for the first time on your machine. At that stage, it tried to **automatically fill the best options** that correspond to your setup. Although tested on several various machine configurations, the auto-configuration script might however fail on your setup. The paragraph below details the possible options that you may want to adjust in the `magic.cfg` file.

9.2.1 Detailed options

In case, the file `magic.cfg` doesn't exist in the directory `$MAGIC_HOME/python/magic`, you can easily copy it from the default configuration `magic.cfg.default` and then adjust the options manually:

```
$ cp $MAGIC_HOME/python/magic/magic.cfg.default $MAGIC_HOME/python/magic/magic.cfg
```

In that file, you can set up the default matplotlib rendering backend (among the possible options: `TkAgg`, `GTKAgg`, `Qt4Agg`, ...). The default configuration is

```
backend = TkAgg
```

Note: This is usually the default configuration which is the most likely to work on supercomputing clusters.

If LaTeX is installed on your work station, you might also want to make use of the better looking LaTeX fonts for all your displayed matplotlib figures (labels, caption, ticks, etc.). Be careful though that most of the time LaTeX is **not installed** on supercomputers. The default configuration is thus:

```
labTex = False
```

If you want to enable all the features of the python functions (faster reading the *G_#.TAG*, conversion to the VTK/VTS file format, potential extrapolation of the field lines, etc.), some fortran libraries present in the *$MAGIC_HOME/python/magic/fortranLib* directory need to be built using the f2py, which should be available on your Linux workstation if all the required python libraries have been correctly installed. The boolean `buildLib` can control whether you want to try building the fortran libraries with f2py. The following configuration will try to build the libraries:

```
buildLib = True
```

The exact name of the executable f2py however varies from one Linux distribution to the other. Among possible options, one frequently finds: `f2py`, `f2py2`, `f2py3`. This can be set to your proper configuration using the `f2pyexec` option of the `magic.cfg` file. The default configuration is:

```
f2pyexec = f2py2
```

You can also choose the fortran compiler you want to use on your machine. A list of the installed compilers can be obtained by using (where `f2py` has to be replaced by your own executable):

```
$ f2py -c --help-fcompiler
```

The most frequent options are:

- `gnu95` for the GNU gfortran compiler.
- `intelem` for the Intel ifort compiler.
- `pg` for the Portlang group pgf compiler.

Once you've decided the ideal configuration for your machine, set it up via the option `fcompiler`:

```
fcompiler = intelem
```

Finally, he same configuration procedure can be applied to the C compiler using the variable named `ccompiler`. The possible options are:

- `unix` for the GNU gcc compiler.
- `intelem` for the Intel icc compiler.

In most of the configurations, the default configuration should do a good job:

```
ccompiler = unix
```

If you encounter any problem during the building stage, you can try playing with this parameter though.

9.2.2 Ready?!

Once you think you set up your `magic.cfg` file correctly, you can test your configuration. If you decided to build the fortran libraries (i.e. `buildLib=True`), you can easily test it with any python shell by typing the following command:

```
>>> from magic import *
```

If the build was successful, it should display:

```
Please wait: building greader_single...
Please wait: building greader_double...
Please wait: building potential extrapolation...
Please wait: building vtklib...
```

Once the libraries have been successfully installed, this message won't be displayed again, except if you remove the `*.so` files that are now present in the `$MAGIC_HOME/python/magic/` directory.

9.3 Python functions and classes

Once the python environment is correctly configured you can use the available functions and classes to analyse and post-process your data. The following pages will give you the detailed API of the available classes, as well as some practical examples:

Python classes

1. To read the **log.TAG** files , see *here*.
2. To read and analyse the time series, see *here*.
3. To read and analyse the radial profiles, see *here*.
4. To read and analyse spectra **_spec_#.TAG**, see *here*.
5. To read and analyse the **G_#.TAG** files, see *here*.
6. To read and analyse movie files **_mov.TAG**, see *here*.
7. To read and analyse coeff files **_coeff.TAG**, see *here*.
8. To read and analyse coeff files **_coeff.TAG**, see *here*.
9. To read and analyse radial spectra **B[r|p]Spec.TAG**, see *here*.
10. To compare several runs simultaneously, see *here*.
11. To transform the graphic files **G_#.TAG** to a file format readable by paraview, VisIt or mayavi and do some fancy 3-D visualisation, see *here*.
12. For additional diagnostics (boundary layer, heat transport, interpolation on cylindrical grids, etc.), see *here*.
13. To take a look at the additional useful functions available (derivation, integration, interpolation, etc.), see *here*.

9.3.1 Support for the `log.TAG` files

class `magic.`**`MagicSetup`**(*datadir='.'*, *nml='input.nml'*, *quiet=False*)

This class allows to read the input namelist or the log file of a current job and creates an object that contains all the parameters found in the namelist/log file.

```
>>> stp = MagicSetup(nml='log.test', quiet=True)
>>> print(stp.ra) # print the Rayleigh number
>>> print(stp.n_r_max) # print n_r_max
```

__init__ (*datadir='.'*, *nml='input.nml'*, *quiet=False*)

Parameters

- **datadir** (*str*) – the working directory
- **nml** (*str*) – name of the input namelist/ log file
- **quiet** (*bool*) – when set to True, makes the output silent (default False)

9.3.2 Support for the time series

class magic.**MagicTs** (*datadir='.'*, *field='e_kin'*, *iplot=True*, *all=False*, *tag=None*)

This python class is used to read and plot the different time series written by the code:

•Kinetic energy: *e_kin.TAG*

•Magnetic energy of the outer core: *e_mag_oc.TAG*

•Magnetic energy of the inner core: *e_mag_ic.TAG*

•Dipole information: *dipole.TAG*

•Rotation: *rot.TAG*

•Diagnostic parameters: *par.TAG*

•Miscellaneous: *misc.TAG*

•Velocity square: *u_square.TAG*

•Angular momentum: *AM.TAG*

•Power budget: *power.TAG*

•Parallel and perpendicular decomposition: *perpPar.TAG*

•RMS force balance: *dtVrms.TAG*

•RMS induction terms: *dtBrms.TAG*

Here are a couple of examples of how to use this function.

```
>>> # plot the most recent e_kin.TAG file found in the directoy
>>> MagicTs(field='e_kin')
>>>
>>> # stack **all** the power.TAG file found in the directory
>>> ts = MagicTs(field='power', all=True)
>>> print(ts.time, ts.buoPower) # print time and buoyancy power
>>>
>>> # If you only want to read the file ``misc.N0m2z``
>>> ts = MagicTs(field='misc', tag='N0m2z', iplot=False)
```

__init__ (*datadir='.'*, *field='e_kin'*, *iplot=True*, *all=False*, *tag=None*)

Parameters

- **field** (*str*) – the file you want to plot
- **iplot** (*bool*) – when set to True, display the plots (default True)
- **all** (*bool*) – when set to True, the complete time series is reconstructed by stacking all the corresponding files from the working directory (default False)
- **tag** (*str*) – read the time series that exactly corresponds to the specified tag

plot()
Plotting subroutines. Only called if 'iplot=True'

9.3.3 Averaging the time series

class magic.**AvgField**(*tstart=None*, *tag=None*, *dipExtra=False*)
This class calculates the time-average properties from time series. It will store the input starting time in a small file named `tInitAvg`, such that the next time you use it you don't need to give `tstart` again.

```
>>> # Average from t=2.11 and also store the additional dipole.TAG informations
>>> a = AvgField(tstart=2.11, dipExtra=True)
>>> # Average only the files that match the pattern N0m2[a-c]
>>> a = AvgField(tstart=2.11, tag='N0m2[a-c]')
>>> # Average only the files that match the pattern N0m2Z*
>>> a = AvgField(tstart=2.11, tag='N0m2Z*')
>>> print(a) # print the formatted output
```

__init__(*tstart=None*, *tag=None*, *dipExtra=False*)

Parameters

- **tstart** (*float*) – the starting time for averaging
- **tag** (*str*) – if you specify an input tag (generic regExp pattern), the averaging process will only happen on the time series that match this input pattern
- **dipExtra** (*bool*) – if this parameter is set to `True`, then additional values extracted from *dipole.TAG* are also computed

__str__()
Formatted output

9.3.4 Support for time-averaged radial profiles

class magic.**MagicRadial**(*datadir='.'*, *field='eKin'*, *iplot=True*, *tag=None*, *tags=None*)
This class can be used to read and display the time and horizontally averaged files:

•Kinetic energy: *eKinR.TAG*

•Magnetic energy: *eMagR.TAG*

•Anelastic reference state: *anel.TAG*

•Variable electrical conductivity: *varCond.TAG*

•Variable thermal diffusivity: *varDiff.TAG*

•Variable kinematic viscosity: *varVisc.TAG*

•Diagnostic parameters: *parR.TAG*

•Power budget: *powerR.TAG*

•Heat fluxes: *fluxesR.TAG*

•Radial profiles used for boundary layers: *bLayersR.TAG*

•Parallel/perpendicular decomposition: *perpParR.TAG*

```
>>> rad = MagicRadial(field='eKinR') # display the content of eKinR.tag
>>> print(rad.radius, rad.ekin_pol_axi) # print radius and poloidal energy
```

__init__ (*datadir='.'*, *field='eKin'*, *iplot=True*, *tag=None*, *tags=None*)

Parameters

- **field** (*str*) – the field you want to plot
- **iplot** (*bool*) – to plot the output, default is True
- **tag** (*str*) – a specific tag, default is None
- **tags** (*list*) – a list that contains multiple tags: useful to sum several radial files

plot()
Display the result when `iplot=True`

9.3.5 Support for the spectra files `(kin|mag|u2)_spec_#.TAG`

class magic.**MagicSpectrum**(*datadir='.'*, *field='e_kin'*, *iplot=True*, *ispec=None*, *ave=False*, *gather=False*, *tag=None*)
This class can be used to read and display the spectra:

•Kinetic energy spectra: *kin_spec_#.TAG*

•Magnetic energy spectra: *mag_spec_#.TAG*

•Spectra of the velocity square: *u2_spec_#.TAG*

```
>>> # display the content of kin_spec_1.tag
>>> # where tag is the most recent file in the current directory
>>> sp = MagicSpectrum(field='e_kin', ispec=1)
>>> # display the content of mag_spec_ave.test on one single figure
>>> sp = MagicSpectrum(field='e_mag', tag='test', ave=True, gather=True)
```

__init__ (*datadir='.'*, *field='e_kin'*, *iplot=True*, *ispec=None*, *ave=False*, *gather=False*, *tag=None*)

Parameters

- **field** (*str*) – the spectrum you want to plot, 'e_kin' for kinetic energy, 'e_mag' for magnetic
- **iplot** (*bool*) – display the output plot when set to True (default is True)
- **ispec** (*int*) – the number of the spectrum you want to plot
- **tag** (*str*) – file suffix (tag), if not specified the most recent one in the current directory is chosen
- **ave** (*bool*) – plot a time-averaged spectrum when set to True
- **gather** (*bool*) – gather the spectra on the same figure when set to True, display one figure per spectrum when set to False, (default is False)
- **datadir** (*str*) – current working directory

plot()
Plotting function

9.3.6 Support for the 2-D spectra files

class magic.**MagicSpectrum2D**(*datadir='.'*, *field='e_mag'*, *iplot=True*, *ispec=None*, *tag=None*, *cm='jet'*, *levels=33*, *precision='Float64'*)
This class can be used to read and display 2-D spectra in the (r, ℓ) and in the (r, m) planes

•Kinetic energy spectra: *2D_kin_spec_#.TAG*

•Velocity square spectra: *2D_u2_spec_#.TAG*

•Magnetic energy spectra: *2D_mag_spec_#.TAG*

```
>>> # display the content of 2D_kin_spec_1.tag
>>> # where tag is the most recent file in the current directory
>>> sp = MagicSpectrum2D(field='e_kin', ispec=1, levels=17, cm='seismic')
>>> # display the content of 2D_mag_spec_3.test
>>> sp = MagicSpectrum2D(field='e_mag', tag='test', ispec=3)
```

__init__ (*datadir='.'*, *field='e_mag'*, *iplot=True*, *ispec=None*, *tag=None*, *cm='jet'*, *levels=33*, *precision='Float64'*)

Parameters

- **field** (*str*) – the spectrum you want to plot, 'e_kin' for kinetic energy, 'e_mag' for magnetic
- **iplot** (*bool*) – display the output when set to True (default is True)
- **ispec** (*int*) – the number of the spectrum you want to plot
- **tag** (*str*) – file suffix (tag=, if not specified the most recent one in the current directory is chosen
- **cm** (*str*) – name of the colormap (default='jet')
- **levels** (*int*) – number of contour levels (default 33)
- **precision** (*str*) – single or double precision
- **datadir** (*str*) – current working directory

plot (*levels*, *cm*)

Plotting function

Parameters

- **levels** (*int*) – number of contour levels
- **cm** (*str*) – name of the colormap

9.3.7 Support for G_#.TAG files

class magic.**MagicGraph** (*ivar=None*, *datadir='.'*, *format='B'*, *quiet=True*, *ave=False*, *tag=None*, *precision='Float32'*)

This class allows to read the 3-D graphic outputs of the MagIC code (*G_#.TAG* and G_ave.TAG) files. Those are binary unformatted outputs, there are therefore two ways to load them:

•If buildLib=True in magic.cfg and the fortran libraries were correctly built, then the reader uses a fortran program that is expected to be much faster than the pure python routine.

•If buildLib=False, then a pure python program is used to read the G files.

```
>>> # Regular G files
>>> gr = MagicGraph(ivar=1, tag='N0m2a')
>>> print(gr.vr.shape) # shape of vr
>>> print(gr.ek) # print ekman number
>>> print(gr.minc) # azimuthal symmetry
>>> # Averaged G file with double precision
>>> gr = MagicGraph(ave=True, tag='N0m2', precision='Float64')
```

__init__ (*ivar=None*, *datadir='.'*, *format='B'*, *quiet=True*, *ave=False*, *tag=None*, *precision='Float32'*)

Parameters

- **format** (*str*) – format of binary output: 'n' (native), 'B' (big endian) or 'l' (little endian), (default 'B')
- **ave** (*bool*) – when set to True, it tries to find an average G file (G_ave.TAG)
- **ivar** (*int*) – the number of the G file
- **tag** (*str*) – extension TAG of the G file. If not specified, the most recent G_#.TAG file found in the directory will be selected.
- **quiet** (*bool*) – when set to True, makes the output silent
- **datadir** (*str*) – directory of the G file (default is .)
- **precision** (*str*) – single or double precision (default 'Float32')

rearangeLat (*field*)

This function is used to unfold the colatitudes

Parameters **field** (*numpy.ndarray*) – input array with MagIC ordering of colatitudes (i.e. successively Northern Hemisphere and Southern Hemisphere)

Returns an array with the regular ordering of the colatitudes

Return type numpy.ndarray

class magic.**Surf** (*ivar=None*, *datadir='.'*, *vort=False*, *ave=False*, *tag=None*, *precision='Float32'*)

This class allows to display the content of a graphic file (*G_#.TAG* or G_ave.TAG). It allows to plot radial, azimuthal and equatorial cuts as well as phi-averages.

```
>>> # To read G_1.test
>>> s = Surf(ivar=1, ave=False, tag='test')
>>> # To read the latest G file in the working directory (double precision)
>>> s = Surf(precision='Float64')
```

```
>>> # Possible plots
>>> s.equat(field='vr')
>>> s.avg(field='vp')
>>> s.surf(field='entropy', r=0.8)
>>> s.slice(field='Br', lon_0=[0, 30])
```

__init__ (*ivar=None*, *datadir='.'*, *vort=False*, *ave=False*, *tag=None*, *precision='Float32'*)

Parameters

- **ivar** (*int*) – index of the graphic file
- **ave** (*bool*) – when set to True, it tries to read a time-averaged graphic file
- **tag** (*str*) – TAG suffix extension of the graphic file
- **vort** (*bool*) – a boolean to specify whether one wants to compute the 3-D vorticiy components (take care of the memory imprint)
- **datadir** (*str*) – the working directory
- **precision** (*str*) – the storage precision of the graphic file (single or double precision). Default is 'Float32' (single)

avg (*field='vphi'*, *levels=16*, *cm='RdYlBu_r'*, *normed=True*, *vmax=None*, *vmin=None*, *cbar=True*, *tit=True*, *pol=False*, *tor=False*, *mer=False*, *merLevels=16*, *polLevels=16*)
Plot the azimutal average of a given field.

```
>>> s = Surf()
>>> # Axisymmetric zonal flows, 65 contour levels
>>> s.avg(field='vp', levels=65, cm='seismic')
```

```
>>> # Minimal plot (no cbar, not title)
>>> s.avg(field='Br', tit=False, cbar=False)
```

```
>>> # Axisymmetric Bphi + poloidal field lines
>>> s.avg(field='Bp', pol=True, polLevels=8)
```

```
>>> # Omega-effect, contours truncated from -1e3 to 1e3
>>> s.avg(field='omeffect', vmax=1e3, vmin=-1e3)
```

Parameters

- **field** (*str*) – the field you want to display
- **levels** (*int*) – the number of levels in the contourf plot
- **cm** (*str*) – name of the colormap ('jet', 'seismic', 'RdYlBu_r', etc.)
- **tit** (*bool*) – display the title of the figure when set to True
- **cbar** (*bool*) – display the colorbar when set to True
- **vmax** (*float*) – maximum value of the contour levels
- **vmin** (*float*) – minimum value of the contour levels
- **normed** (*bool*) – when set to True, the colormap is centered around zero. Default is True, except for entropy/temperature plots.
- **pol** (*bool*) – diplay the poloidal field lines contours when set to Tru
- **mer** (*bool*) – display the meridional circulation contours when set to True
- **merLevels** (*int*) – number of contour levels to display meridional circulation
- **polLevels** (*int*) – number of contour levels to display poloidal field lines

equat (*field='vr'*, *levels=16*, *cm='RdYlBu_r'*, *normed=True*, *vmax=None*, *vmin=None*, *cbar=True*, *tit=True*, *avg=False*, *normRad=False*)
Plot the equatorial cut of a given field

```
>>> s = Surf()
>>> # Equatorial cut of the z-vorticity, 65 contour levels
>>> s.equat(field='vortz', levels=65, cm='seismic')
```

```
>>> # Minimal plot (no cbar, not title)
>>> s.equat(field='bphi', tit=False, cbar=False)
```

```
>>> # Control the limit of the colormap from -1e3 to 1e3
>>> s.equat(field='vr', vmin=-1e3, vmax=1e3, levels=33)
```

```
>>> # Normalise the contour levels radius by radius
>>> s.equat(field='jphi', normRad=True)
```

Parameters

- **field** (*str*) – the name of the input physical quantity you want to display
- **avg** (*bool*) – when set to True, an additional figure which shows the radial profile of the input physical quantity (azimuthal average) is also displayed
- **normRad** (*bool*) – when set to True, the contour levels are normalised radius by radius (default is False)
- **levels** (*int*) – the number of levels in the contour
- **cm** (*str*) – name of the colormap ('jet', 'seismic', 'RdYlBu_r', etc.)
- **tit** (*bool*) – display the title of the figure when set to True
- **cbar** (*bool*) – display the colorbar when set to True
- **vmax** (*float*) – maximum value of the contour levels
- **vmin** (*float*) – minimum value of the contour levels
- **normed** (*bool*) – when set to True, the colormap is centered around zero. Default is True, except for entropy/temperature plots.

slice (*field='Bphi'*, *lon_0=0.0*, *levels=12*, *cm='RdYlBu_r'*, *normed=True*, *vmin=None*, *vmax=None*, *cbar=True*, *tit=True*, *grid=False*, *nGridLevs=16*)

Plot an azimuthal slice of a given field.

```
>>> s = Surf()
>>> # vphi at 0, 30, 60 degrees in longitude
>>> s.slice(field='vp', lon_0=[0, 30, 60], levels=65, cm='seismic')
```

```
>>> # Minimal plot (no cbar, not title)
>>> s.avg(field='vp', lon_0=32, tit=False, cbar=False)
```

```
>>> # Axisymmetric Bphi + poloidal field lines
>>> s.avg(field='Bp', pol=True, polLevels=8)
```

```
>>> # Omega-effect, contours truncated from -1e3 to 1e3
>>> s.avg(field='omeffect', vmax=1e3, vmin=-1e3)
```

Parameters

- **field** (*str*) – the field you want to display
- **lon_0** (*float or list*) – the longitude of the slice in degrees, or a list of longitudes
- **levels** (*int*) – the number of levels in the contourf plot
- **cm** (*str*) – name of the colormap ('jet', 'seismic', 'RdYlBu_r', etc.)
- **tit** (*bool*) – display the title of the figure when set to True
- **cbar** (*bool*) – display the colorbar when set to True
- **vmax** (*float*) – maximum value of the contour levels
- **vmin** (*float*) – minimum value of the contour levels
- **grid** (*bool*) – display or hide the grid
- **nGridLevs** (*int*) – number of grid levels

surf (*field='Bphi'*, *proj='hammer'*, *lon_0=0.0*, *r=0.85*, *vmax=None*, *vmin=None*, *lat_0=30.0*, *levels=16*, *cm='RdYlBu_r'*, *normed=True*, *cbar=True*, *tit=True*, *lines=False*)

Plot the surface distribution of an input field at a given input radius (normalised by the outer boundary radius).

```
>>> s = Surf()
>>> # Radial flow component at ``r=0.95 r_o``, 65 contour levels
>>> s.surf(field='vr', r=0.95, levels=65, cm='seismic')
```

```
>>> # Minimal plot (no cbar, not title)
>>> s.surf(field='entropyfluct', r=0.6, tit=False, cbar=False)
```

```
>>> # Control the limit of the colormap from -1e3 to 1e3
>>> s.surf(field='vp', r=1., vmin=-1e3, vmax=1e3, levels=33)
```

```
>>> # If basemap is installed, additional projections are available
>>> s.surf(field='Br', r=0.95, proj='ortho', lat_0=45, lon_0=45)
```

Parameters

- **field** (*str*) – the name of the field you want to display
- **proj** (*str*) – the type of projection. Default is Hammer, in case you want to use 'ortho' or 'moll', then Basemap is required.
- **r** (*float*) – the radius at which you want to display the input data (in normalised units with the radius of the outer boundary)
- **levels** (*int*) – the number of levels in the contour
- **cm** (*str*) – name of the colormap ('jet', 'seismic', 'RdYlBu_r', etc.)
- **tit** (*bool*) – display the title of the figure when set to True
- **cbar** (*bool*) – display the colorbar when set to True
- **lines** (*bool*) – when set to True, over-plot solid lines to highlight the limits between two adjacent contour levels
- **vmax** (*float*) – maximum value of the contour levels
- **vmin** (*float*) – minimum value of the contour levels
- **normed** (*bool*) – when set to True, the colormap is centered around zero. Default is True, except for entropy/temperature plots.

9.3.8 Support for movie files `*_mov.TAG`

class magic.**Movie** (*file=None*, *iplot=True*, *step=1*, *png=False*, *lastvar=None*, *nvar='all'*, *levels=12*, *cm='RdYlBu_r'*, *cut=0.5*, *bgcolor=None*, *fluct=False*, *normed=False*, *avg=False*, *std=False*, *dpi=80*, *normRad=False*, *precision='Float32'*)

This class allows to read the *movie files* generated by the MagIC code.

```
>>> m = Movie()
>>> # This returns a list of the available movies in the directory
>>> # and lets you decide which one you want to read
```

```
>>> # Reads and display AV_mov.test
>>> m = Movie(filed='AV_mov.test')
>>> print(m.data) # access to the data
```

```
>>> # Read three movie files (no display)
>>> m1 = Movie(file='AV_mov.testa', iplot=True)
>>> m2 = Movie(file='AV_mov.testb', iplot=True)
>>> m3 = Movie(file='AV_mov.testc', iplot=True)
>>> # Stack them together
>>> m = m1+m2+m3
>>> # Display
>>> m.plot(levels=33, cm='seismic', cut=0.5)
```

```
>>> # Store the outputs in movie/img_#.png
>>> # Only from the timesteps 280 to 380
>>> m = Movie(file='AB_mov.test', png=True, nvar=100, lastvar=380)
```

__add__ (*new*)
Built-in function to sum two movies

Note: So far this function only works for two movies with the same grid sizes. At some point, we might introduce grid extrapolation to allow any summation/

__init__ (*file=None*, *iplot=True*, *step=1*, *png=False*, *lastvar=None*, *nvar='all'*, *levels=12*, *cm='RdYlBu_r'*, *cut=0.5*, *bgcolor=None*, *fluct=False*, *normed=False*, *avg=False*, *std=False*, *dpi=80*, *normRad=False*, *precision='Float32'*)

Parameters

- **nvar** (*int*) – the number of timesteps of the movie file we want to plot starting from the last line
- **png** (*bool*) – if png=True, write the png files instead of display
- **iplot** (*bool*) – if iplot=True, display otherwise just read
- **lastvar** (*int*) – the number of the last timesteps to be read
- **step** (*int*) – the stepping between two timesteps
- **levels** (*int*) – the number of contour levels
- **cm** (*str*) – the name of the color map
- **fluct** (*bool*) – if fluct=True, substract the axisymmetric part
- **normed** (*bool*) – the colormap is rescaled every timestep when set to True, otherwise it is calculated from the global extrema
- **avg** (*bool*) – if avg=True, time-average is displayed
- **std** (*bool*) – if std=True, standard deviation is displayed
- **dpi** (*int*) – dot per inch when saving PNGs
- **normRad** (*bool*) – if normRad=True, then we normalise for each radial level
- **precision** (*str*) – precision of the input file, Float32 for single precision, Float64 for double precision
- **cut** (*float*) – adjust the contour extrema to max(abs(data))*cut
- **bgcolor** (*str*) – background color of the figure

avgStd (*std=False*, *cut=0.5*, *levels=12*, *cmap='RdYlBu_r'*)
plot time-average or standard deviation

Parameters

- **std** (*bool*) – the standard deviation is computed instead the average when std is True
- **levels** (*int*) – number of contour levels
- **cmap** (*str*) – name of the colormap
- **cut** (*float*) – adjust the contour extrema to max(abs(data))*cut

plot(*cut=0.5*, *levels=12*, *cmap='RdYlBu_r'*, *png=False*, *step=1*, *normed=False*, *dpi=80*, *bgcolor=None*)
Plotting function (it can also write the png files)

Parameters

- **levels** (*int*) – number of contour levels
- **cmap** (*str*) – name of the colormap
- **cut** (*float*) – adjust the contour extrema to max(abs(data))*cut
- **png** (*bool*) – save the movie as a series of png files when set to True
- **dpi** (*int*) – dot per inch when saving PNGs
- **bgcolor** (*str*) – background color of the figure
- **normed** (*bool*) – the colormap is rescaled every timestep when set to True, otherwise it is calculated from the global extrema
- **step** (*int*) – the stepping between two timesteps

class magic.**Movie3D**(*file=None*, *step=1*, *lastvar=None*, *nvar='all'*, *nrout=48*, *ratio_out=2.0*, *potExtra=False*, *precision='Float32'*)
This class allows to read the 3D movie files *(B|V)_3D_.TAG* and transform them into a series of VTS files `./vtsFiles/B3D_#.TAG` that can be further read using paraview.

```
>>> Movie3D(file='B_3D.TAG')
```

__init__(*file=None*, *step=1*, *lastvar=None*, *nvar='all'*, *nrout=48*, *ratio_out=2.0*, *potExtra=False*, *precision='Float32'*)

Parameters

- **file** (*str*) – file name
- **nvar** (*int*) – the number of timesteps of the movie file we want to plot starting from the last line
- **lastvar** (*int*) – the number of the last timestep to be read
- **step** (*int*) – the stepping between two timesteps
- **precision** (*str*) – precision of the input file, Float32 for single precision, Float64 for double precision
- **potExtra** (*bool*) – when set to True, potential extrapolation of the magnetic field outside the fluid domain is also computed
- **ratio_out** (*float*) – ratio of desired external radius to the CMB radius. This is is only used when potExtra=True
- **nrout** (*int*) – number of additional radial grid points to compute the potential extrapolation. This is only used when potExtra=True

9.3.9 Support for `B_cmb_coeff.TAG` and `(V|B)_coeff_r#.TAG` files

class `magic.coeff.`**`MagicCoeffCmb`**(*tag*, *ratio_cmb_surface=1*, *scale_b=1*, *iplot=True*, *precision='Float64'*)

This class allows to read the *B_coeff_cmb.TAG* files. It first read the poloidal potential at the CMB and then transform it to the Gauss coefficients $g_{\ell m}$ and $h_{\ell m}$ using the getGauss function.

```
>>> # Reads the files B_coeff_cmb.testa, B_coeff_cmb.testb
>>> # and B_coeff_cmb.testc and stack them in one single time series
>>> cmb = MagicCoeffCmb(tag='test[a-c]')
>>> print(cmb.ell, cmb.glm) # print \ell and g_{\ell m}
>>> print(cmb.glm[:, 1, 0]) # time-series of the axisymmetric dipole
>>> plot(cmb.time, cmb.dglmdt[:, 1, 0]) # Secular variation of the dipole
```

`__init__`(*tag*, *ratio_cmb_surface=1*, *scale_b=1*, *iplot=True*, *precision='Float64'*)

A class to read the B_coeff_cmb files

Parameters

- **`tag`** (*str*) – if you specify a pattern, it tries to read the corresponding files
- **`ratio_cmb_surface`** (*float*) – ratio of surface ratio to CMB radius (default is 1)
- **`scale_b`** (*float*) – magnetic field unit (default is 1)
- **`iplot`** (*int*) – a logical to toggle the plot (default is True)
- **`precision`** (*char*) – single or double precision

`plot`()

Display some results when iplot is set to True

class `magic.coeff.`**`MagicCoeffR`**(*tag*, *ratio_cmb_surface=1*, *scale_b=1*, *iplot=True*, *field='B'*, *r=1*, *precision='Float64'*)

This class allows to read the *B_coeff_r#.TAG* and *V_coeff_r#.TAG* files. It reads the poloidal and toroidal potentials and reconstruct the time series (or the energy) contained in any given mode.

```
>>> # Reads the files V_coeff_r2.test*
>>> cr = MagicCoeffR(tag='test*', field='V', r=2)
>>> print(cr.ell, cr.wlm) # print \ell and w_{\ell m}
>>> # Time-evolution of the poloidal energy in the (\ell=10, m=10) mode
>>> plot(cr.time, cr.epolLM[:, 10, 10])
```

`__init__`(*tag*, *ratio_cmb_surface=1*, *scale_b=1*, *iplot=True*, *field='B'*, *r=1*, *precision='Float64'*)

Parameters

- **`tag`** (*str*) – if you specify a pattern, it tries to read the corresponding files
- **`ratio_cmb_surface`** (*float*) – ratio of surface ratio to CMB radius (default is 1)
- **`scale_b`** (*float*) – magnetic field unit (default is 1)
- **`iplot`** (*bool*) – a logical to toggle the plot (default is True)
- **`field`** (*str*) – 'B', 'V' or 'T' (magnetic field, velocity field or temperature)
- **`r`** (*int*) – an integer to characterise which file we want to plot
- **`precision`** (*str*) – single or double precision

`magic.coeff.`**`deriv`**(*x*, *y*, *axis=0*)

This function is a simple second order derivative

Parameters

- **x** (*numpy.ndarray*) – input x-axis
- **y** (*numpy.ndarray*) – input array

Returns an array that contains the derivatives

Return type numpy.ndarray

magic.coeff.**getGauss**(*alm*, *blm*, *ell*, *m*, *scale_b*, *ratio_cmb_surface*, *rcmb*)
Get the Gauss coefficients from the real and imaginary parts of the poloidal potential

Parameters

- **alm** (*numpy.ndarray*) – real part of the poloidal potential
- **blm** (*numpy.ndarray*) – imaginary part of the poloidal potential
- **ell** (*numpy.ndarray*) – spherical harmonic degree ell
- **scale_b** (*float*) – magnetic field unit (default is 1)
- **ratio_cmb_surface** (*float*) – ratio of surface ratio to CMB radius (default is 1)
- **rcmb** (*float*) – radius of the outer boundary

9.3.10 Support for B[rp]Spec.TAG

class magic.**MagicRSpec**(*tag*, *field='Br'*, *precision='Float32'*, *avg=False*)
This class allows to read the *rB[r|p]Spec.TAG files*. Those files contain the time-evolution of the poloidal/toroidal magnetic energy for all radii and for spherical harmonic degrees from 1 to 6. This is an unformatted fortran file.

```
>>> # Read all the `BrSpec.test*` files in the current working directory and
>>> # stack them.
>>> rsp = MagicRSpec(tag='test*', field='Br')
```

__init__(*tag*, *field='Br'*, *precision='Float32'*, *avg=False*)

Parameters

- **tag** (*str*) – if you specify a pattern, it tries to read the corresponding files and stack them.
- **field** (*str*) – nature of the radial spectra. Possible choices are 'Bt' or 'Bp'
- **precision** (*str*) – single or double precision (default single, i.e. 'Float32')
- **avg** (*bool*) – when set to True, display time averaged quantities

plotAvg()
Plotting function for time-averaged profiles

9.3.11 Support for TO outputs

class magic.**TOMovie**(*file=None*, *iplot=True*, *cm='RdYlBu_r'*, *cut=0.8*, *levels=16*, *avg=True*, *precision='Float32'*)
This class allows to read and display the *TO_mov.TAG* generated when *l_TOmovie=.true.* is True.

```
>>> # This will allow you to pick up one TO_mov files among the existing ones
>>> t = TOMovie()
```

```
>>> # Read TO_mov.N0m2, time-averaged it and display it with 65 contour levels
>>> t = TOMovie(file='TO_mov.N0m2', avg=True, levels=65, cm='seismic')
```

__init__ (*file=None*, *iplot=True*, *cm='RdYlBu_r'*, *cut=0.8*, *levels=16*, *avg=True*, *precision='Float32'*)

Parameters

- **file** (*str*) – the filename of the TO_mov file
- **cmap** (*str*) – the name of the color map
- **levels** (*int*) – the number of contour levels
- **cut** (*float*) – adjust the contour extrema to max(abs(data))*cut
- **iplot** (*bool*) – a boolean to specify if one wants to plot or not the results
- **avg** (*bool*) – time average of the different forces
- **precision** (*str*) – precision of the input file, Float32 for single precision, Float64 for double precision

plot (*cut=0.8*, *levs=16*, *avg=True*, *cmap='RdYlBu_r'*)
Plotting function

Parameters

- **cut** (*float*) – adjust the contour extrema to max(abs(data))*cut
- **levs** (*int*) – number of contour levels
- **avg** (*bool*) – when set to True, quantities are time-averaged
- **cmap** (*str*) – name of the colormap

9.3.12 Run comparison

class magic.**CompSims** (*file='liste'*, *field='ts'*, *ncol=4*, *cm='RdYlBu_r'*, *dpi=96*, *normed=True*, *levels=16*, *type=None*, *r=0.9*, *bw=False*, *ave=False*, *cut=1*)
This class allows to compare an analyse several DNS simultaneously. It is possible to compare time-series or *graphic files*. To set it up, you first need to create a file that contains the list of directories you want to analyse:

```
$ cat inputList
E3e4Eps5e3Q05
E3e4Eps2e3Q07
E3e4Eps2e3Q08
E3e4Eps2e3Q09
```

This list thus contains four directories (one run per directory) that can be further analysed:

```
>>> # Display the time-series of kinetic energy on 2 columns
>>> CompSims(file='inputList', field='ts', ncol=2)
>>> # Display the equatorial cuts of v_r
>>> CompSims(file='inputList', field='vr', type='equat', levels=65, cm='seismic')
>>> # Display the radial cuts of B_r at r=0.8 r_o
>>> CompSims(file='inputList', field='br', type='surf', r=0.8)
>>> # Display the average zonal flow
>>> CompSims(file='inputList', field='vp', type='avg')
```

__init__ (*file='liste'*, *field='ts'*, *ncol=4*, *cm='RdYlBu_r'*, *dpi=96*, *normed=True*, *levels=16*, *type=None*, *r=0.9*, *bw=False*, *ave=False*, *cut=1*)

Parameters

- **file** (*str*) – the input file that contains the list of directories that one wants to analyse

- **field** (*str*) – name of the input field. Possible options are: 'ts': displaye the time-series of kinetic energy; 'e_mag': display the time-series of magnetic energy; 'flux': display the time-series of the Nusselt numbers; 'zonal': display the surface zonal flow; 'Anything else': try to interpret the field
- **type** (*str*) – nature of the plot. Possible values are: 'avg' or 'slice': phi-average or phi-slice; 'equat': equatorial cut; 'surf': radial cut; 'ts*: time series
- **ncol** (*int*) – number of columns of the figure
- **ave** (*bool*) – when set to True, it tries to read a time-averaged graphic file
- **r** (*float*) – the radius at which you want to display the input data (in normalised units with the radius of the outer boundary)
- **levels** (*int*) – the number of levels in the contour
- **cm** (*str*) – name of the colormap ('jet', 'seismic', 'RdYlBu_r', etc.)
- **normed** (*bool*) – when set to True, the colormap is centered around zero. Default is True, except for entropy/temperature plots.
- **dpi** (*int*) – dot per inch when saving PNGs
- **bw** (*bool*) – when set to True, display grey-scaled contour levels
- **cut** (*float*) – adjust the contour extrema to max(abs(data))*cut

plotAvg()
Plot azimutal averages in (theta, r) planes.

plotEmag()
Plot time-series of the magnetic energy

plotEquat()
Plot equatorial cuts in (phi, r) planes.

plotFlux()
Plot time-series of the top and bottom Nusselt numbers

plotSurf()
Plot radial cuts in (phi, theta) planes using the Hammer projection.

plotTs()
Plot time-series of the kinetic energy

plotZonal()
Plot surface zonal flow profiles.

9.3.13 Conversion of G_#.TAG files to vts/vti files

class magic.graph2vtk.**Graph2Vtk**(*gr, filename='out', scals=['vr', 'emag', 'tfluct'], vecs=['u', 'B'], potExtra=False, ratio_out=2, nrout=32, deminc=True, outType='vts', nFiles=1, nx=96, ny=96, nz=96*)

This class allows to transform an input graphic file to a file format readable by paraview/visit or mayavi. It also allows to compute a possible potential extrapolation of the field lines in an arbitrary outer spherical shell domain

```
>>> # Load a graphic file
>>> gr = MagicGraph(ivar=1)
>>> # store myOut.vts
>>> Graph2Vtk(gr, 'myOut', outType='vts')
>>> # store u' and B for the vector fields and vortz and T for the scalars
```

```
>>> Graph2Vtk(gr, scals=['temp', 'vortz'], vecs=['ufluct', 'B'])
>>> # store only T'
>>> Graph2Vtk(gr, scals=['tempfluct'], vecs=[])
>>> # store only B with its potential extrapolation up to 3*r_cmb
>>> Graph2Vtk(gr, scals=[], vecs=['B'], potExtra=True, ratio_out=3)
>>> # Extrapolate on a cartesian grid of size 128^3
>>> Graph2Vtk(gr, outType='vti', nx=128, ny=128, nz=128)
```

__init__ (*gr, filename='out', scals=['vr', 'emag', 'tfluct'], vecs=['u', 'B'], potExtra=False, ratio_out=2, nrout=32, deminc=True, outType='vts', nFiles=1, nx=96, ny=96, nz=96*)

Parameters

- **filename** (*str*) – the file name of the output (without extension)
- **gr** (magic.MagicGraph) – the input graphic file one wants to transform to vts/vti
- **scals** (*list(str)*) – a list that contains the possible input scalars: 'entropy', 'vr', 'vp', 'tfluct', 'vortz', 'vortzfluct', 'ekin', 'emag', 'vortr'
- **vecs** (*list(str)*) – a list that contains the possible input vectors: 'u', 'b', 'ufluct', 'bfluct'
- **potExtra** (*bool*) – when set to True, calculates the potential extrapolation of the magnetic field up to ratio_out*r_cmb
- **ratio_out** (*float*) – in case of potential extrapolation, this is the ratio of the external outer radius to r_cmb (rout/rcmb)
- **nrout** (*integer*) – in case of potential extrapolation, this input allows to specify thenumber of radial grid points in the outer spherical envelope
- **deminc** (*bool*) – a logical to indicate if one wants do get rid of the possible azimuthal symmetry
- **outType** (*str*) – nature of the VTK file produced. This can be either 'vts' for the spherical grid or 'vti' for an extrapolation on a cartesian grid
- **nFiles** (*int*) – number of output chunks in case of parallel vts file format (pvts)
- **nx** (*int*) – number of grid points in the x direction
- **ny** (*int*) – number of grid points in the x direction
- **nz** (*int*) – number of grid points in the x direction

writeVTI (*filename, nx=96, ny=96, nz=96*)

In this case, the output is extrapolated on a cartesian grid and then written in a vti file.

Parameters

- **filename** (*str*) – the file name of the output (without extension)
- **nx** (*int*) – number of grid points in the x direction
- **ny** (*int*) – number of grid points in the x direction
- **nz** (*int*) – number of grid points in the x direction

writeVTS (*filename, nFiles*)

This function stores the output on a structured-grid vts file.

Parameters

- **filename** (*str*) – the file name of the output (without extension)
- **nFiles** (*int*) – number of outpute files (in case of pvts)

magic.graph2vtk.**sph2cart_scal**(*scals*, *radius*, *nx=96*, *ny=96*, *nz=96*, *minc=1*)
This function interpolates a series of scalar fields from the spherical coordinates to the cartesian coordinates.

Parameters

- **scals** (*numpy.ndarray[nscals,nphi,ntheta,nr]*) – an array that contains the different scalar quantities
- **radius** (*numpy.ndarray*) – the input radius
- **nx** (*int*) – number of grid points in the x direction
- **ny** (*int*) – number of grid points in the x direction
- **nz** (*int*) – number of grid points in the x direction
- **minc** (*int*) – azimuthal symmetry

Returns a tuple that contains the scalars, the max of the grid and the grid spacing

Return type (numpy.ndarray[nscals,nz,ny,nx],float,float)

magic.graph2vtk.**sph2cart_vec**(*vecr*, *vect*, *vecp*, *radius*, *nx=96*, *ny=96*, *nz=96*, *minc=1*)
This function interpolates a series of vector fields from the spherical coordinates to the cartesian coordinates.

Parameters

- **vecr** (*numpy.ndarray[nvecs,nphi,ntheta,nr]*) – the radial components of the different vector fields
- **vect** (*numpy.ndarray[nvecs,nphi,ntheta,nr]*) – the latitudinal components of the different vector fields
- **vecp** (*numpy.ndarray[nvecs,nphi,ntheta,nr]*) – the azimuthal components of the different vector fields
- **radius** (*numpy.ndarray*) – the input radius
- **nx** (*int*) – number of grid points in the x direction
- **ny** (*int*) – number of grid points in the x direction
- **nz** (*int*) – number of grid points in the x direction
- **minc** (*int*) – azimuthal symmetry

Returns a tuple that contains the three vectors components

Return type (numpy.ndarray[nvecs,nz,ny,nx],...)

9.3.14 Potential extrapolation

class magic.potExtra.**ExtraPot**(*rcmb*, *brcmb*, *minc*, *ratio_out=2.0*, *nrout=32*, *cutCMB=False*, *deminc=True*)
This class is used to compute the potential field extrapolation of the magnetic field in an arbitrary outer spherical shell domain. It takes as an input the magnetic field at the CMB.

__init__(*rcmb*, *brcmb*, *minc*, *ratio_out=2.0*, *nrout=32*, *cutCMB=False*, *deminc=True*)

Parameters

- **bcmb** (*numpy.ndarary*) – the surface radial field, array of dimension [np, nt]
- **rcmb** (*float*) – the value of the radius at the surface
- **minc** (*int*) – azimuthal symmetry

- **ratio_out** (*float*) – the ratio of the outer sphere radius to the surface radius
- **nrout** (*int*) – the number of radial point (linearly spaced) of the extrapolated field in the outer spherical domain
- **cutCMB** (*bool*) – a logical if one wants to remove the first grid point (useful if one then wants to merge the graphic file with the extrapolation)
- **deminc** (*bool*) – a logical to indicate if one wants do get rid of the possible azimuthal symmetry

avg (*field='br'*, *levels=12*, *cm='RdYlBu_r'*, *normed=True*, *vmax=None*, *vmin=None*)
A small routine to plot the azimuthal averages of the extrapolated fields.

Parameters

- **field** (*str*) – the quantity you want to plot: 'br' or 'bp'
- **levels** (*int*) – the number of contour levels
- **cm** (*str*) – the name of the colormap
- **vmax** (*float*) – maximum value of the contour levels
- **vmin** (*float*) – minimum value of the contour levels
- **normed** (*bool*) – when set to True, the colormap is centered around zero. Default is True, except for entropy/temperature plots.

9.3.15 Additional possible analyses

class magic.bLayers.**BLayers**(*iplot=False*, *quiet=False*)
This class allows to determine the viscous and thermal boundary layers using several classical methods (slope method, peak values, dissipation rates, etc.). It uses the following files:

•Kinetic energy: *eKinR.TAG*

•Power budget: *powerR.TAG*

•Radial profiles used for boundary layers: *bLayersR.TAG*

This function can thus **only** be used when both *powerR.TAG* and *bLayersR.TAG* exist in the working directory.

> **Warning:** This function works well as long as rigid boundaries and fixed temperature boundary conditions are employed. Other combination of boundary conditions (fixed fluxes and/or stress-free) might give wrong results, since boundary layers become awkward to define in that case.

Since this function is supposed to use time-averaged quantities, the usual procedure is first to define the initial averaging time using `AvgField`: (this needs to be done only once)

```
>>> a = AvgField(tstart=2.58)
```

Once the `tInitAvg` file exists, the boundary layer calculation can be done:

```
>>> bl = BLayers(iplot=True)
>>> # print the formatted output
>>> print(bl)
```

__init__(*iplot=False*, *quiet=False*)

Parameters

- **iplot** (*bool*) – display the result when set to True (default False)

- **quiet** (*bool*) – less verbose when set to True (default is False)

__str__()
Formatted output

plot()
Plotting function

magic.bLayers.**getAccuratePeaks**(*rad*, *uh*, *uhTop*, *uhBot*, *ri*, *ro*)
This functions performs a spline extrapolation around the maxima of the input array uh to define a more accurate location of the boundary layer.

Parameters

- **rad** (*numpy.ndarray*) – radius
- **uh** (*numpy.ndarray*) – the horizontal velocity profile
- **uhTop** (*float*) – first peak value of uh close to the outer boundary
- **uhBot** (*float*) – first peak value of uh close to the inner boundary
- **ri** (*float*) – the inner core radius
- **ro** (*float*) – the outer core radius

Returns four floats: thickness of the bottom boundary layer, thickness of the top boundary layer, extrapolated value of uh at the bottom boundary layer, extrapolated value of uh at the top boundary layer

Return type list

magic.bLayers.**getMaxima**(*field*)
This function determines the local maxima of the input array field

Parameters **field** (*numpy.ndarray*) – the input array

Returns a list containing the indices of the local maxima

Return type list

magic.bLayers.**integBotTop**(*rad*, *field*, *ri*, *ro*, *lambdai*, *lambdao*, *normed=False*)
This function evaluates the radial integral of the input array field in the bottom and top boundary layers separately.

Parameters

- **rad** (*numpy.ndarray*) – radius
- **field** (*numpy.ndarray*) – the input radial profile
- **ri** (*float*) – the inner core radius
- **ro** (*float*) – the outer core radius
- **lambdai** (*float*) – thickness of the inner boundary layer
- **lambdao** (*float*) – thickness of the outer boundary layer
- **normed** (*bool*) – when set to True, the outputs are normalised by the volumes of the boundary layers. In that case, the outputs are volume-averaged quantities.

Returns two floats that contains the bottom and top boundary layers integrations (integBot, integTop)

Return type list

magic.bLayers.**integBulkBc**(*rad*, *field*, *ri*, *ro*, *lambdai*, *lambdao*, *normed=False*)

This function evaluates the radial integral of the input array field in the boundary layer and in the bulk separately.

Parameters

- **rad** (*numpy.ndarray*) – radius
- **field** (*numpy.ndarray*) – the input radial profile
- **ri** (*float*) – the inner core radius
- **ro** (*float*) – the outer core radius
- **lambdai** (*float*) – thickness of the inner boundary layer
- **lambdao** (*float*) – thickness of the outer boundary layer
- **normed** (*bool*) – when set to True, the outputs are normalised by the volumes of the boundary layers and the fluid bulk, respectively. In that case, the outputs are volume-averaged quantities.

Returns two floats that contains the boundary layer and the bulk integrations (integBc, integBulk)

Return type list

class magic.**ThetaHeat**(*iplot=False*, *angle=10*, *pickleName='thHeat.pickle'*)

This class allows to conduct some analysis of the latitudinal variation of the heat transfer. It relies on the movie files *ATmov.TAG* and *AHF_mov.TAG*. As it's a bit time-consuming, the calculations are stored in a python.pickle file to quicken future usage of the data.

This function can **only** be used when *bLayersR.TAG* exist in the working directory.

Since this function is supposed to use time-averaged quantities, the usual procedure is first to define the initial averaging time using *AvgField*: (this needs to be done only once)

```
>>> a = AvgField(tstart=2.58)
```

Once the tInitAvg file exists, the latitudinal heat transfer analysis can be done using:

```
>>> # For chunk-averages over 10^\degree in the polar and equatorial regions.
>>> th = ThetaHeat(angle=10)
>>> # Formatted output
>>> print(th)
```

__init__(*iplot=False*, *angle=10*, *pickleName='thHeat.pickle'*)

Parameters

- **iplot** (*bool*) – a boolean to toggle the plots on/off
- **angle** (*float*) – the integration angle in degrees

PickleName calculations a

__str__()

Formatted outputs

```
>>> th = ThetaHeat()
>>> print(th)
```

plot()

Plotting function

class magic.cyl.**Cyl**(*ivar=1*, *datadir='.'*, *ns=None*)

This class allows to extrapolate a given *graphic file* on a cylindrical grid. Once done, the extrapolated file is

stored in a python.pickle file. It is then possible to display 2-D cuts of the extrapolated arrays (radial cuts, phi-averages, equatorial cuts, z-averages and phi-slices)

> **Warning:** This process is actually **very demanding** and it might take a lot of time to extrapolate the G_#.TAG file. Be careful when choosing the input value of ns!

```
>>> # Extrapolate the G file to the cylindrical grid (ns=128, nz=2*ns)
>>> c = Cyl(ivar=1, ns=128)
>>> # Radial cut of v_r
>>> c.surf(field='vr', r=0.8)
>>> # Vertical average of B_\phi
>>> c.avgz(field='Bphi', cm='seismic', levels=33)
>>> # Azimuthal average of v_\phi
>>> c.avg(field='Bphi')
>>> # Equatorial cut of of v_theta
>>> c.equat(field='vtheta')
```

__init__ (*ivar=1*, *datadir='.'*, *ns=None*)

Parameters

- **ivar** (*int*) – the number of the Graphic file
- **datadir** (*str*) – working directory
- **ns** (*int*) – number of grid points in the radial direction

avg (*field='Bphi'*, *levels=16*, *cm='RdYlBu_r'*, *normed=True*, *vmax=None*, *vmin=None*)
Plot the azimutal average of a given field.

```
>>> c = Cyl(ns=65)
>>> # Azimuthal average of B_r
>>> c.avg(field='Br', cm='seismic', levels=33)
```

Parameters

- **field** (*str*) – name of the input field
- **levels** (*int*) – number of contour levels
- **cm** (*str*) – name of the color map
- **normed** (*bool*) – when set to True, the contours are normalised fro -max(field), max(field)
- **vmin** (*float*) – truncate the contour levels to values > vmin
- **vmax** (*float*) – truncate the contour levels to values < vmax

avgz (*field='vs'*, *levels=16*, *cm='RdYlBu_r'*, *normed=True*, *vmin=None*, *vmax=None*, *avg=False*)
Plot the vertical average of a given field.

```
>>> c = Cyl(ns=65)
>>> # Vertical average of v_s
>>> c.avg(field='vs', cm='seismic', levels=33)
```

Parameters

- **field** (*str*) – name of the input field
- **levels** (*int*) – number of contour levels
- **cm** (*str*) – name of the color map

- **normed** (*bool*) – when set to True, the contours are normalised fro -max(field), max(field)
- **vmin** (*float*) – truncate the contour levels to values > vmin
- **vmax** (*float*) – truncate the contour levels to values < vmax
- **avg** (*bool*) – when set to True, an additional figure with the phi-average profile is also displayed

equat (*field='vs'*, *levels=16*, *cm='RdYlBu_r'*, *normed=True*, *vmax=None*, *vmin=None*)
Plot an input field in the equatorial plane.

```
>>> c = Cyl(ns=65)
>>> # Equatorial cut of v_\phi
>>> c.equat(field='vphi', cm='seismic', levels=33)
```

Parameters

- **field** (*str*) – name of the input field
- **levels** (*int*) – number of contour levels
- **cm** (*str*) – name of the color map
- **normed** (*bool*) – when set to True, the contours are normalised fro -max(field), max(field)
- **vmin** (*float*) – truncate the contour levels to values > vmin
- **vmax** (*float*) – truncate the contour levels to values < vmax

slice (*field='Bphi'*, *lon_0=0.0*, *levels=16*, *cm='RdYlBu_r'*, *normed=True*)
Plot an azimuthal slice of a given field.

```
>>> c = Cyl(ns=65)
>>> # Slices of v_r at 30 and 60 degrees
>>> c.slice(field='vr', lon_0=[30, 60])
```

Parameters

- **field** (*str*) – name of the input field
- **lon_0** (*float or list*) – the longitude of the slice in degrees, or a list of longitudes
- **levels** (*int*) – number of contour levels
- **cm** (*str*) – name of the color map
- **normed** (*bool*) – when set to True, the contours are normalised fro -max(field), max(field)

surf (*field='Bphi'*, *r=0.85*, *vmin=None*, *vmax=None*, *levels=16*, *cm='RdYlBu_r'*, *normed=True*, *figsize=None*)
Plot the surface distribution of an input field at a given input radius (normalised by the outer boundary radius).

```
>>> c = Cyl(ns=65)
>>> # Surface plot of B_\phi from -10 to 10
>>> c.surf(field='Bphi', r=0.6, vmin=-10, vmax=10, levels=65)
```

Parameters

- **field** (*str*) – name of the input field
- **r** (*float*) – radial level (normalised to the outer boundary radius)

- **levels** (*int*) – number of contour levels
- **cm** (*str*) – name of the color map
- **normed** (*bool*) – when set to True, the contours are normalised fro -max(field), max(field)
- **vmin** (*float*) – truncate the contour levels to values > vmin
- **vmax** (*float*) – truncate the contour levels to values < vmax

magic.cyl.**sph2cyl**(*g*, *ns=None*, *nz=None*)

This function interpolates the three flow (or magnetic field) component of a *G_#.TAG* file on a cylindrical grid of size (ns, nz).

> **Warning:** This might be really slow!

Parameters

- **g** (*magic.MagicGraph*) – input graphic output file
- **ns** (*int*) – number of grid points in the radial direction
- **nz** (*int*) – number of grid points in the vertical direction

Returns a python tuple of five numpy.ndarray (S,Z,vs,vp_cyl,vz). S[nz,ns] is a meshgrid that contains the radial coordinate. Z[nz,ns] is a meshgrid that contains the vertical coordinate. vs[nz,ns] is the radial component of the velocity (or magnetic field), vp_cyl[nz,ns] the azimuthal component and vz[nz,ns] the vertical component.

Return type tuple

magic.cyl.**sph2cyl_plane**(*data*, *rad*, *ns*, *nz*)

This function extrapolates a phi-slice of a spherical shell on a cylindrical grid

```
>>> # Read G_1.test
>>> gr = MagicGraph(ivar=1, tag='test')
>>> # phi-average v_\phi and s
>>> vpm = gr.vphi.mean(axis=0)
>>> sm = gr.entropy.mean(axis=0)
>>> # Interpolate on a cylindrical grid
>>> S, Z, outputs = sph2cyl_plane([vpm, sm], gr.radius, 512, 1024)
>>> vpm_cyl, sm_cyl = outputs
```

Parameters

- **data** (*list(numpy.ndarray)*) – a list of 2-D arrays [(ntheta, nr), (ntheta, nr), ...]
- **rad** (*numpy.ndarray*) – radius
- **ns** (*int*) – number of grid points in s direction
- **nz** (*int*) – number of grid points in z direction

Returns a python tuple that contains two numpy.ndarray and a list (S,Z,output). S[nz,ns] is a meshgrid that contains the radial coordinate. Z[nz,ns] is a meshgrid that contains the vertical coordinate. output=[arr1[nz,ns], ..., arrN[nz,ns]] is a list of the interpolated array on the cylindrical grid.

Return type tuple

magic.cyl.**zavg**(*input*, *radius*, *ns*, *minc*, *save=True*, *filename='vp.pickle'*, *normed=True*)

This function computes a z-integration of a list of input arrays (on the spherical grid). This works well for 2-D (phi-slice) arrays. In case of 3-D arrays, only one element is allowed (too demanding otherwise).

Parameters

- **input** (*list(numpy.ndarray)*) – a list of 2-D or 3-D arrays
- **radius** (*numpy.ndarray*) – spherical radius
- **ns** (*int*) – radial resolution of the cylindrical grid (nz=2*ns)
- **minc** (*int*) – azimuthal symmetry
- **save** (*bool*) – a boolean to specify if one wants to save the outputs into a pickle (default is True)
- **filename** (*str*) – name of the output pickle when save=True
- **normed** (*bool*) – a boolean to specify if ones wants to simply integrate over z or compute a z-average (default is True: average)

Returns a python tuple that contains two numpy.ndarray and a list (height,cylRad,output) height[ns] is the height of the spherical shell for all radii. cylRad[ns] is the cylindrical radius. output=[arr1[ns], ..., arrN[ns]] contains the z-integrated output arrays.

Return type tuple

class magic.**Butterfly**(*file=None*, *step=1*, *iplot=True*, *rad=0.8*, *lastvar=None*, *nvar='all'*, *levels=20*, *cm='RdYlBu_r'*, *precision='Float32'*, *cut=0.8*)

This class can be used to display the time evolution of the magnetic field for various latitudes (i.e. the well-known butterfly diagrams). These diagrams are usually constructed using MagIC's *movie files*: either radial cuts (like Br_CMB_mov.TAG) or azimuthal-average (like AB_mov.TAG).

```
>>> # Read Br_CMB_mov.ccondAnelN3MagRa2e7Pm2ggg
>>> t1 = Butterfly(file='Br_CMB_mov.ccondAnelN3MagRa2e7Pm2ggg', step=1,
                   iplot=False)
>>> # Plot it
>>> t1.plot(levels=33, cm='seismic', cut=0.6)
```

__add__(*new*)

Overload of the addition operator

```
>>> # Read 2 files
>>> b1 = Butterfly(file='AB_mov.test1', iplot=False)
>>> b2 = Butterfly(file='AB_mov.test2', iplot=False)
>>> # Stack them and display the whole thing
>>> b = b1+b2
>>> b.plot(levels=33, contour=True, cut=0.8, cm='seismic')
```

__init__(*file=None*, *step=1*, *iplot=True*, *rad=0.8*, *lastvar=None*, *nvar='all'*, *levels=20*, *cm='RdYlBu_r'*, *precision='Float32'*, *cut=0.8*)

Parameters

- **file** (*str*) – when specified, the constructor reads this file, otherwise a list with the possible options is displayed
- **rad** (*float*) – radial level (normalised to the outer boundary radius)
- **iplot** (*bool*) – display/hide the plots (default is True)
- **nvar** (*int*) – the number of time steps (lines) of the movie file we want to plot starting from the last line
- **lastvar** (*int*) – the number of the last time step to be read
- **step** (*int*) – the stepping between two lines

- **levels** (*int*) – the number of contour levels
- **cm** (*str*) – the name of the color map
- **cut** (*float*) – adjust the contour extrema to max(abs(data))*cut
- **precision** (*bool*) – precision of the input file, Float32 for single precision, Float64 for double precision

fourier2D (*renorm=False*)

This function allows to conduct some basic Fourier analysis on the data. It displays two figures: the first one is a contour levels in the (Frequency, Latitude) plane, the second one is integrated over latitudes (thus a simple, power vs Frequency plot)

```
>>> # Load the data without plotting
>>> b1 = Butterfly(file='AB_mov.test1', iplot=False)
>>> # Fourier analysis
>>> b1.fourier2D()
```

Parameters **renorm** (*bool*) – when set to True, it rebins the time series in case of irregularly spaced data

plot (*levels=12*, *contour=False*, *renorm=False*, *cut=0.5*, *mesh=3*, *cm='RdYlBu_R'*)

Plotting function

Parameters

- **cm** (*str*) – name of the colormap
- **levels** (*int*) – the number of contour levels (only used when iplot=True and contour=True)
- **contour** (*bool*) – when set to True, display contour levels (pylab.contourf), when set to False, display an image (pylab.imshow)
- **renorm** (*bool*) – when set to True, it re-bins the time series in case of irregularly time-spaced data
- **mesh** (*int*) – when renorm=True, factor of regriding: NewTime = mesh*OldTime
- **cut** (*float*) – adjust the contour extrema to max(abs(data))*cut

9.3.16 Various useful functions

magic.libmagic.**anelprof** (*radius*, *strat*, *polind*, *g0=0.0*, *g1=0.0*, *g2=1.0*)

This functions calculates the reference temperature and density profiles of an anelastic model.

```
>>> rad = chebgrid(65, 1.5, 2.5)
>>> temp, rho, beta = anelprof(rad, strat=5., polind=2.)
```

Parameters

- **radius** (*numpy.ndarray*) – the radial gridpoints
- **polind** (*float*) – the polytropic index
- **strat** (*float*) – the number of the density scale heights between the inner and the outer boundary
- **g0** (*float*) – gravity profile: g=g0
- **g1** (*float*) – gravity profile: g=g1*r/r_o

- **g2** (*float*) – gravity profile: g=g2*(r_o/r)**2

Returns a tuple that contains the temperature profile, the density profile and the log-derivative of the density profile versus radius

Return type (numpy.ndarray, numpy.ndarray, numpy.ndarray)

magic.libmagic.**avgField**(*time*, *field*, *tstart*)

This subroutine computes the time-average of a time series

```
>>> ts = MagicTs(field='misc', iplot=False, all=True)
>>> nuavg = avgField(ts.time, ts.topnuss, 0.35)
>>> print(nuavg)
```

Parameters

- **time** (*numpy.ndarray*) – time
- **field** (*numpy.ndarray*) – the time series of a given field
- **tstart** (*float*) – the starting time of the averaging

Returns the time-averaged quantity

Return type float

magic.libmagic.**chebgrid**(*nr*, *a*, *b*)

This function defines a Gauss-Lobatto grid from a to b.

```
>>> r_icb = 0.5 ; r_cmb = 1.5; n_r_max=65
>>> rr = chebgrid(n_r_max, r_icb, r_cmb)
```

Parameters

- **nr** (*int*) – number of radial grid points
- **a** (*float*) – lower limit of the Gauss-Lobatto grid
- **b** (*float*) – upper limit of the Gauss-Lobatto grid

Returns the Gauss-Lobatto grid

Return type numpy.ndarray

magic.libmagic.**cut**(*dat*, *vmax=None*, *vmin=None*)

This functions truncates the values of an input array that are beyond vmax or below vmin and replace them by vmax and vmin, respectively.

```
>>> # Keep only values between -1e3 and 1e3
>>> datNew = cut(dat, vmin=-1e3, vmax=1e3)
```

Parameters

- **dat** (*numpy.ndarray*) – an input array
- **vmax** (*float*) – maximum upper bound
- **vmin** (*float*) – minimum lower bound

Returns an array where the values >=vmax have been replaced by vmax and the values <=vmin have been replaced by vmin

Return type numpy.ndarray

magic.libmagic.**cylSder**(*radius*, *data*)
This function computes the s derivative of an input array defined on a regularly-spaced cylindrical grid.

```
>>> s = linspace(0., 1., 129 ; dat = cos(s)
>>> ddatds = cylSder(s, dat)
```

Parameters

- **radius** (*numpy.ndarray*) – cylindrical radius
- **data** (*numpy.ndarray*) – input data

Returns s derivative

Return type numpy.ndarray

magic.libmagic.**cylZder**(*z*, *data*)
This function computes the z derivative of an input array defined on a regularly-spaced cylindrical grid.

```
>>> z = linspace(-1., 1., 129 ; dat = cos(z)
>>> ddatdz = cylZder(z, dat)
```

Parameters

- **z** (*numpy.ndarray*) – height of the cylinder
- **data** (*numpy.ndarray*) – input data

Returns z derivative

Return type numpy.ndarray

magic.libmagic.**fast_read**(*file*, *skiplines=0*, *binary=False*, *precision='Float64'*)
This function reads an input ascii table (can read both formatted or unformatted fortran)

```
>>> # Read 'e_kin.test', skip the first 10 lines
>>> data = fast_read('e_kin.test', skiplines=10)
```

Parameters

- **file** (*str*) – name of the input file
- **skiplines** (*int*) – number of header lines to be skept during reading
- **binary** (*bool*) – when set to True, try to read an unformatted binray Fortran file (default is False)
- **precision** (*str*) – single ('Float32') or double precision ('Float64')

Returns an array[nlines, ncols] that contains the data of the ascii file

Return type numpy.ndarray

magic.libmagic.**getCpuTime**(*file*)
This function calculates the CPU time from one given log file

Parameters **file** (*file*) – the log file you want to analyze

Returns the total CPU time

Return type float

magic.libmagic.**getTotalRunTime**()
This function calculates the total CPU time of one run directory

Returns the total RUN time

Return type float

magic.libmagic.**hammer2cart**(*ttheta*, *pphi*, *colat=False*)
This function is used to define the Hammer projection used when plotting surface contours in *magic.Surf*

```
>>> # Load Graphic file
>>> gr = MagicGraph()
>>> # Meshgrid
>>> pphi, ttheta = mgrid[-N.pi:N.pi:gr.nphi*1j, N.pi/2.:-N.pi/2.:gr.ntheta*1j]
>>> x,y = hammer2cart(ttheta, pphi)
>>> # Contour plots
>>> contourf(x, y, gr.vphi)
```

Parameters

- **ttheta** (*numpy.ndarray*) – meshgrid [nphi, ntheta] for the latitudinal direction
- **pphi** – meshgrid [nphi, ntheta] for the azimuthal direction
- **colat** (*numpy.ndarray*) – colatitudes (when not specified a regular grid is assumed)

Returns a tuple that contains two [nphi, ntheta] arrays: the x, y meshgrid used in contour plots

Return type (numpy.ndarray, numpy.ndarray)

magic.libmagic.**intcheb**(*f*, *nr*, *z1*, *z2*)
This function integrates an input function f defined on the Gauss-Lobatto grid.

```
>>> print(intcheb(f, 65, 0.5, 1.5))
```

Parameters

- **f** – an input array
- **nr** (*int*) – number of radial grid points
- **z1** (*float*) – lower limit of the Gauss-Lobatto grid
- **z2** (*float*) – upper limit of the Gauss-Lobatto grid

Type numpy.ndarray

Returns the integrated quantity

Return type float

magic.libmagic.**matder**(*nr*, *z1*, *z2*)
This function calculates the derivative in Chebyshev space.

```
>>> r_icb = 0.5 ; r_cmb = 1.5; n_r_max=65
>>> d1 = matder(n_r_max, r_icb, r_cmb)
>>> # Chebyshev grid and data
>>> rr = chebgrid(n_r_max, r_icb, r_cmb)
>>> f = sin(rr)
>>> # Radial derivative
>>> df = dot(d1, f)
```

Parameters

- **nr** (*int*) – number of radial grid points
- **z1** (*float*) – lower limit of the Gauss-Lobatto grid
- **z2** (*float*) – upper limit of the Gauss-Lobatto grid

Returns a matrix of dimension (nr,nr) to calculate the derivatives

Return type numpy.ndarray

magic.libmagic.**phideravg**(*data*, *minc=1*, *order=4*)
phi-derivative of an input array

```
>>> gr = MagicGraph()
>>> dvphidp = phideravg(gr.vphi, minc=gr.minc)
```

Parameters

- **data** (*numpy.ndarray*) – input array
- **minc** (*int*) – azimuthal symmetry
- **order** (*int*) – order of the finite-difference scheme (possible values are 2 or 4)

Returns the phi-derivative of the input array

Return type numpy.ndarray

magic.libmagic.**progressbar**(*it*, *prefix=''*, *size=60*)
Fancy progress-bar for loops

```
for i in progressbar(range(1000000)):
    x = i
```

Parameters

- **prefix** (*str*) – prefix string before progress bar
- **size** (*int*) – width of the progress bar (in points of xterm width)

magic.libmagic.**rderavg**(*data*, *eta=0.35*, *spectral=True*, *exclude=False*)
Radial derivative of an input array

```
>>> gr = MagiGraph()
>>> dvrdr = rderavg(gr.vr, eta=gr.radratio)
```

Parameters

- **data** (*numpy.ndarray*) – input array
- **eta** (*float*) – aspect ratio of the spherical shell
- **spectral** (*bool*) – when set to True use Chebyshev derivatives, otherwise use finite differences (default is True)
- **exclude** (*bool*) – when set to True, exclude the first and last radial grid points and replace them by a spline extrapolation (default is False)

Returns the radial derivative of the input array

Return type numpy.ndarray

magic.libmagic.**scanDir**(*pattern*, *tfix=None*)
This function sorts the files which match a given input pattern from the oldest to the most recent one (in the current working directory)

```
>>> dat = scanDir('log.*')
>>> print(log)
```

Parameters

- **pattern** (*str*) – a classical regexp pattern
- **tfix** (*float*) – in case you want to add only the files that are more recent than a certain date, use tfix (computer 1970 format!!)

Returns a list of files that match the input pattern

Return type list

magic.libmagic.**sderavg**(*data*, *eta=0.35*, *spectral=True*, *colat=None*, *exclude=False*)
s derivative of an input array

```
>>> gr = MagiGraph()
>>> dvpds = sderavg(gr.vphi, eta=gr.radratio, colat=gr.colatitude)
```

Parameters

- **data** (*numpy.ndarray*) – input array
- **eta** (*float*) – aspect ratio of the spherical shell
- **spectral** (*bool*) – when set to True use Chebyshev derivatives, otherwise use finite differences (default is True)
- **exclude** (*bool*) – when set to True, exclude the first and last radial grid points and replace them by a spline extrapolation (default is False)
- **colat** (*numpy.ndarray*) – colatitudes (when not specified a regular grid is assumed)

Returns the s derivative of the input array

Return type numpy.ndarray

magic.libmagic.**selectField**(*obj*, *field*, *labTex=True*)
This function selects for you which field you want to display. It actually allows to avoid possible variables miss-spelling: i.e. 'Bphi'='bp'='Bp'='bphi'

Parameters

- **obj** (*magic.MagicGraph*) – a graphic output file
- **field** (*str*) – the name of the field one wants to select
- **labTex** (*bool*) – when set to True, format the labels using LaTeX fonts

Returns a tuple that contains the selected physical field and its label

Return type (numpy.ndarray, str)

magic.libmagic.**symmetrize**(*data*, *ms*, *reversed=False*)
Symmetrise an array which is defined only with an azimuthal symmetry minc=ms

Parameters

- **data** (*numpy.ndarray*) – the input array

- **ms** (*int*) – the azimuthal symmetry
- **reversed** (*bool*) – set to True, in case the array is reversed (i.e. n_phi is the last column)

Returns an output array of dimension (data.shape[0]*ms+1)

Return type numpy.ndarray

magic.libmagic.**thetaderavg**(*data*, *order=4*)

Theta-derivative of an input array (finite differences)

```
>>> gr = MagiGraph()
>>> dvtdt = thetaderavg(gr.vtheta)
```

Parameters

- **data** (*numpy.ndarray*) – input array
- **order** (*int*) – order of the finite-difference scheme (possible values are 2 or 4)

Returns the theta-derivative of the input array

Return type numpy.ndarray

magic.libmagic.**writeVpEq**(*par*, *tstart*)

This function computes the time-averaged surface zonal flow (and Rolc) and format the output

```
>>> # Reads all the par.* files from the current directory
>>> par = MagicTs(field='par', iplot=False, all=True)
>>> # Time-average
>>> st = writeVpEq(par, tstart=2.1)
>>> print(st)
```

Parameters

- **par** (*magic.MagicTs*) – a *MagicTs* object containing the par file
- **tstart** (*float*) – the starting time of the averaging

Returns a formatted string

Return type str

magic.libmagic.**zderavg**(*data*, *eta=0.35*, *spectral=True*, *colat=None*, *exclude=False*)

z derivative of an input array

```
>>> gr = MagiGraph()
>>> dvrdz = zderavg(gr.vr, eta=gr.radratio, colat=gr.colatitude)
```

Parameters

- **data** (*numpy.ndarray*) – input array
- **eta** (*float*) – aspect ratio of the spherical shell
- **spectral** (*bool*) – when set to True use Chebyshev derivatives, otherwise use finite differences (default is True)
- **exclude** (*bool*) – when set to True, exclude the first and last radial grid points and replace them by a spline extrapolation (default is False)
- **colat** (*numpy.ndarray*) – colatitudes (when not specified a regular grid is assumed)

Returns the z derivative of the input array

Return type numpy.ndarray

CHAPTER

TEN

DESCRIPTION OF THE FORTRAN MODULES

The following pages contain an exhaustive description of the different variables, subroutines and modules used in MagIC. This documentation is automatically generated from the source code docstrings using the Sphinx extention for the Fortran domain.

Fortran modules

1. For the main program file `magic.f90`, see *here*.
2. For the core modules that contain most of the global variables, see *here*.
3. For the MPI related modules, see *here*.
4. For the code initialization and the pre-calculations done in the initial stage of the computation (before the time-stepping loop), see *here* and *there*.
5. For the time-stepping loop, see *here*.
6. For the calculation of the non-linear terms (in the physical space) and their time-advance, see *here*.
7. For the calculation of the linear terms (in spectral space) and their time-advance, see *here*.
8. For the Chebyshev, Fourier and Legendre transforms, see *here*.
9. For the computation of the radial derivatives (Chebyshev) and the integration, see *here*.
10. For the definition of the blocking, see *here*.
11. For the calculation of the standard outputs (time-series, spectra and radial files), see *here*.
12. For the calculation of binary outputs (graphic files, movie files, potential and coeff files), see *here*.
13. For the additional calculations of specific outputs (torsional oscillations, RMS force balance, etc.), see *here*.
14. For reading and writing the check points (restart files), see *here*.
15. For additional useful functions (string manipulation, HDF5 support, etc.), see *here*.

10.1 Main program `magic.f90`

program `magic`

A dynamic dynamo model driven by thermal convection in a rotating spherical fluid shell. This version can solve for both Boussinesq and anelastic fluids and non-dimensional variables are used throughout the whole code.

Use *horizontal_data*, *rms* (*initialize_rms()*), *output_mod* (*initialize_output()*), *movie_data* (*finalize_movie_data()*, *initialize_movie_data()*), *lmloop_mod* (*initialize_lmloop()*), *lmloop_data* (*initialize_lmloop_data()*), *init_fields*, *precision_mod*, *precalculations*, *fields_average_mod*, *magnetic_energy*, *communications* (*initialize_communications()*), *grenoble*, *parallel_mod*, *step_time_mod* (*step_time()*, *initialize_step_time()*), *num_param*, *physical_parameters*, *radial_functions*, *output_data*,

power (*initialize_output_power()*), *namelists*, *outto_mod* (*initialize_outto_mod()*), *torsional_oscillations*, *timing* (*writetime()*, *walltime()*), *kinetic_energy*, *matrices*, *blocking*, *constants* (*codeversion()*), *truncation*, *fieldslast*, *egeos_mod*, *fields*, *spectra* (*initialize_spectra()*), *dtb_mod* (*initialize_dtb_mod()*), *radial_data* (*initialize_radial_data()*), *radialloop*, *outpar_mod* (*initialize_outpar_mod()*), *logic*, *outpv3* (*initialize_outpv3()*), *start_fields* (*getstartfields()*)

Call to *parallel()*, *walltime()*, *writetime()*, *readnamelists()*, *initialize_output()*, *checktruncation()*, *openfiles()*, *initialize_blocking()*, *initialize_radial_data()*, *initialize_radial_functions()*, *initialize_radialloop()*, *initialize_lmloop_data()*, *initialize_lmloop()*, *initialize_num_param()*, *initialize_to()*, *initialize_outto_mod()*, *initialize_init_fields()*, *initialize_grenoble()*, *initialize_horizontal_data()*, *initialize_matrices()*, *initialize_fields()*, *initialize_fieldslast()*, *initialize_dtb_mod()*, *initialize_kinetic_energy()*, *initialize_magnetic_energy()*, *initialize_fields_average_mod()*, *initialize_egeos_mod()*, *initialize_spectra()*, *initialize_outpv3()*, *initialize_step_time()*, *initialize_communications()*, *initialize_outpar_mod()*, *initialize_output_power()*, *precalc()*, *initialize_movie_data()*, *initialize_rms()*, *writenamelists()*, *getstartfields()*, *precalctimes()*, *writeinfo()*, *step_time()*, *finalize_movie_data()*, *closefiles()*

10.2 Base modules

10.2.1 `precision.f90`

Description

This module controls the precision used in MagIC

Quick access

Variables *sizeof_character*, *sizeof_integer*, *mpi_def_complex*, *mpi_out_real*, *mpi_def_real*, *lip*, *cp*, *outp*, *sizeof_def_complex*, *sizeof_out_real*, *sizeof_def_real*

Needed modules

- `mpi`

Variables

- `precision_mod/`**`outp`** *[integer,parameter*=selected_real_kind(6)/public*]*
- `precision_mod/`**`mpi_out_real`** *[integer,parameter*=mpi_real4/public*]*

- precision_mod/**sizeof_integer** *[integer,parameter*=4/public*]*
- precision_mod/**sizeof_character** *[integer,parameter*=1/public*]*
- precision_mod/**mpi_def_complex** *[integer,parameter*=mpi_complex16/public*]*
- precision_mod/**mpi_def_real** *[integer,parameter*=mpi_real8/public*]*
- precision_mod/**sizeof_def_real** *[integer,parameter*=8/public*]*
- precision_mod/**sizeof_out_real** *[integer,parameter*=4/public*]*
- precision_mod/**lip** *[integer,parameter*=selected_int_kind(12)/public*]*
- precision_mod/**cp** *[integer,parameter*=selected_real_kind(15)/public*]*
- precision_mod/**sizeof_def_complex** *[integer,parameter*=16/public*]*

10.2.2 truncation.f90

Description

This module defines the grid points and the truncation

Quick access

Variables *ncp*, *ncpgeos*, *l_maxmag*, *lm_maxmag*, *lgeos*, *lavemem*, *lmagmem*, *ldtbmem*, *lmoviemem*, *lstressmem*, *n_r_ic_max_dtb*, *n_r_ic_max_ave*, *nrpgeos*, *lm_maxgeos*, *lm_max_dtb*, *lmp_max_dtb*, *lm_max_ave*, *n_phi_maxstr*, *n_r_maxgeos*, *n_r_max_dtb*, *n_r_maxstr*, *n_r_max_ave*, *n_r_ic_maxmag*, *n_r_maxmag*, *n_theta_maxstr*, *n_phi_max*, *minc*, *nalias*, *nrp*, *l_max*, *n_cheb_max*, *n_m_max*, *m_max*, *n_r_totmag*, *n_cheb_ic_max*, *n_r_ic_max*, *lm_max_real*, *lmp_max*, *lm_max*, *n_phi_tot*, *n_r_max*, *n_theta_max*, *n_r_tot*

Routines *checktruncation()*, *initialize_truncation()*

Variables

- truncation/**n_r_ic_max** *[integer]*
  number of grid points in inner core
- truncation/**l_max** *[integer]*
  max degree of Plms
- truncation/**minc** *[integer]*
  basic wavenumber, longitude symmetry
- truncation/**n_cheb_ic_max** *[integer]*
  number of chebs in inner core
- truncation/**lstressmem** *[integer]*
  Memory for stress output
- truncation/**lgeos** *[integer]*
  Memory for Geostrophic output
- truncation/**n_r_maxmag** *[integer]*
  Number of radial points to calculate magnetic field

- truncation/**n_phi_maxstr** *[integer]*
  Number of phi points for stress output
- truncation/**n_theta_max** *[integer]*
  number of theta grid-points
- truncation/**lm_maxmag** *[integer]*
  Max. number of l/m combinations for magnetic field calculation
- truncation/**nrp** *[integer]*
  dimension of phi points in for real/complex arrays
- truncation/**n_r_max_ave** *[integer]*
  Number of radial points for time average
- truncation/**ldtbmem** *[integer]*
  Memory for movie output
- truncation/**lm_maxgeos** *[integer]*
  Number of l/m combinations for Geostrophic output
- truncation/**nrpgeos** *[integer]*
  Number of cyl. radial points for Geostrophic output
- truncation/**n_r_max** *[integer]*
  number of radial grid points
- truncation/**n_theta_maxstr** *[integer]*
  Number of theta points for stress output
- truncation/**lmoviemem** *[integer]*
  Memory for movies
- truncation/**nalias** *[integer]*
  controls dealiasing in latitude and
- truncation/**n_r_maxgeos** *[integer]*
  Number of radial points for Geostrophic output
- truncation/**n_r_tot** *[integer]*
  total number of radial grid points
- truncation/**ncp** *[integer]*
- truncation/**lmp_max_dtb** *[integer]*
  Number of l/m combinations for movie output if l runs to l_max+1
- truncation/**lmp_max** *[integer]*
  number of l/m combination if l runs to l_max+1
- truncation/**n_m_max** *[integer]*
  max number of ms (different oders)
- truncation/**m_max** *[integer]*
  max order of Plms
- truncation/**lm_max** *[integer]*
  number of l/m combinations
- truncation/**lavemem** *[integer]*
  Memory for calculating time averages
- truncation/**lm_max_ave** *[integer]*
  Number of l/m combinations for time average

- truncation/**n_r_ic_max_ave** *[integer]*
  Number of IC radial points for time average
- truncation/**n_phi_tot** *[integer]*
  number of longitude grid points
- truncation/**lm_max_real** *[integer]*
  number of l/m combination for real representation (cos/sin)
- truncation/**n_r_ic_maxmag** *[integer]*
  Number of radial points to calculate IC magnetic field
- truncation/**n_r_maxstr** *[integer]*
  Number of radial points for stress output
- truncation/**ncpgeos** *[integer]*
- truncation/**n_phi_max** *[integer]*
  absolute number of phi grid-points
- truncation/**n_r_ic_max_dtb** *[integer]*
  Number of IC radial points for movie output
- truncation/**n_r_max_dtb** *[integer]*
  Number of radial points for movie output
- truncation/**l_maxmag** *[integer]*
  Max. degree for magnetic field calculation
- truncation/**lm_max_dtb** *[integer]*
  Number of l/m combinations for movie output
- truncation/**n_r_totmag** *[integer]*
  n_r_maxMag + n_r_ic_maxMag
- truncation/**n_cheb_max** *[integer]*
  max degree-1 of cheb polynomia
- truncation/**lmagmem** *[integer]*
  Memory for magnetic field calculation

Subroutines and functions

subroutine truncation/**initialize_truncation**()

Called from *readnamelists()*

subroutine truncation/**checktruncation**()
This function checks truncations and writes it into STDOUT and the log-file. MPI: called only by the processor responsible for output !

Called from *magic*

10.2.3 num_param.f90

Description

Module containing numerical and control parameters

Quick access

Variables *n_lscale*, *n_tscale*, *ldif*, *ldifexp*, *n_cour_step*, *n_time_steps*, *istop*, *amstart*, *pscale*, *difeta*, *difkap*, *difnu*, *enscale*, *escale*, *dtstart*, *lscale*, *dtmax*, *runtimelimit*, *dtmin*, *tend*, *timestart*, *runtime*, *tscale*, *intfac*, *alffac*, *courfac*, *vscale*, *runtimestart*, *alpha*, *delxh2*, *delxr2*

Routines *initialize_num_param()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *truncation* (*n_r_max()*): This module defines the grid points and the truncation

Variables

- num_param/**amstart** *[real,public]*
- num_param/**n_time_steps** *[integer,public]*
  Total number of time steps requested in the name list
- num_param/**difnu** *[real,public]*
  Amplitude of viscous hyperdiffusion
- num_param/**timestart** *[real,public]*
  Numerical time where run should start
- num_param/**dtmin** *[real,public]*
  Minimum allowed time step
- num_param/**delxr2** (:) *[real,allocatable/public]*
- num_param/**ldifexp** *[integer,public]*
  Exponent for hyperdiffusion function
- num_param/**difkap** *[real,public]*
  Amplitude of thermal hyperdiffusion
- num_param/**dtstart** *[real,public]*
  Initial time step if start solution is initialized
- num_param/**n_lscale** *[integer,public]*
  Control length scale
- num_param/**tend** *[real,public]*
  Numerical time where run should end
- num_param/**runtimelimit** (4) *[integer,public]*
  Maximum running time
- num_param/**istop** *[integer,public]*
  Variable used in FFT soubroutine
- num_param/**ldif** *[integer,public]*
  Degree where hyperdiffusion starts to act
- num_param/**alffac** *[real,public]*
  Value to scale Alfen-velocity in courant criteria

- num_param/**vscale** *[real,public]*
  Velocity scale
- num_param/**escale** *[real,public]*
  Energy scale
- num_param/**delxh2** (:) *[real,allocatable/public]*
- num_param/**courfac** *[real,public]*
  Value to scale velocity in courant criteria
- num_param/**tscale** *[real,public]*
  Time scale
- num_param/**dtmax** *[real,public]*
  Maximum allowed time step
- num_param/**alpha** *[real,public]*
  Weight for implicit time step
- num_param/**n_tscale** *[integer,public]*
  Control time scale
- num_param/**pscale** *[real,public]*
- num_param/**runtimestart** (4) *[integer,public]*
  Wall clock time of start of the run
- num_param/**difeta** *[real,public]*
  Amplitude of magnetic hyperdiffusion
- num_param/**enscale** *[real,public]*
  Energies scale
- num_param/**n_cour_step** *[integer,public]*
  Step for controlling Courant criteria
- num_param/**intfac** *[real,public]*
  Value to re-scale dtMax during simulation
- num_param/**runtime** (4) *[integer,public]*
  Running time
- num_param/**lscale** *[real,public]*
  Length scale

Subroutines and functions

subroutine num_param/**initialize_num_param**()

Called from *magic*

10.2.4 phys_param.f90

Description

Module containing the physical parameters

Quick access

Variables *kbotb*, *kbots*, *kbotv*, *ktops*, *imps*, *n_imps*, *imagcon*, *ktopb*, *mode*, *n_r_lcr*, *nvareps*, *nvarcond*, *nvardiff*, *nvarvisc*, *ktopv*, *interior_model*, *ekscaled*, *rascaled*, *epss*, *dissnb*, *ek*, *con_lambdamatch*, *epsc0*, *lffac*, *opr*, *corfac*, *opm*, *o_sr*, *conductance_ma*, *r_cut_model*, *pr*, *ohmlossfac*, *vischeatfac*, *r_lcr*, *buofac*, *ra*, *epsc*, *rho_ratio_ic*, *rho_ratio_ma*, *g2*, *g0*, *g1*, *con_decrate*, *difexp*, *tmagcon*, *con_radratio*, *sigma_ratio*, *radratio*, *prmag*, *con_funcwidth*, *con_lambdaout*, *strat*, *polind*, *cmbhflux*, *slopestrat*, *n_imps_max*, *peaks*, *phis*, *thetas*, *widths*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC

Variables

- physical_parameters/**n_imps** *[integer]*
  Heat boundary condition
- physical_parameters/**lffac** *[real]*
  Inverse of Pr*Ekman
- physical_parameters/**ek** *[real]*
  Ekman number
- physical_parameters/**ktopb** *[integer]*
  Magnetic boundary condition
- physical_parameters/**strat** *[real]*
  number of density scale heights
- physical_parameters/**polind** *[real]*
  polytropic index
- physical_parameters/**epsc0** *[real]*
  Internal heat source magnitude
- physical_parameters/**vischeatfac** *[real]*
  Prefactor for viscous heating: $Di\,Pr/Ra$
- physical_parameters/**ktopv** *[integer]*
  Velocity boundary condition
- physical_parameters/**con_decrate** *[real]*
  Slope of electrical conductivity profile (nVarCond=2)
- physical_parameters/**dissnb** *[real]*
  Dissipation number
- physical_parameters/**ktops** *[integer]*
  Entropy boundary condition
- physical_parameters/**opm** *[real]*
  Inverse of magnetic Prandtl number
- physical_parameters/**rho_ratio_ic** *[real]*
  Same density as outer core

- physical_parameters/**conductance_ma** *[real]*
  OC conductivity
- physical_parameters/**peaks** (20) *[real]*
- physical_parameters/**pr** *[real]*
  Prandtl number
- physical_parameters/**con_lambdamatch** *[real]*
  Electrical conductivity at con_RadRatio (nVarCond=2)
- physical_parameters/**cmbhflux** *[real]*
  stratified Layer
- physical_parameters/**sigma_ratio** *[real]*
  Value of IC rotation
- physical_parameters/**epsc** *[real]*
  Renormalisation of epsc0
- physical_parameters/**r_cut_model** *[real]*
  Percentage on the inner part of the interior model to be used
- physical_parameters/**ra** *[real]*
  Rayleigh number
- physical_parameters/**n_r_lcr** *[integer]*
  Number of radial points where conductivity is zero
- physical_parameters/**o_sr** *[real]*
  Inverse of sigma_ratio
- physical_parameters/**rho_ratio_ma** *[real]*
  Same density as outer core
- physical_parameters/**corfac** *[real]*
  Inverse of ekScaled
- physical_parameters/**ohmlossfac** *[real]*
  Prefactor for Ohmic heating: $Di\,Pr/(Ra\,E\,Pm^2)$
- physical_parameters/**mode** *[integer]*
  Mode of calculation
- physical_parameters/**thetas** (20) *[real]*
- physical_parameters/**buofac** *[real]*
  Ratio of Rayleigh number over Prandtl number
- physical_parameters/**kbots** *[integer]*
- physical_parameters/**g2** *[real]*
  Set to 1.0 for $1/r^2$ gravity
- physical_parameters/**g1** *[real]*
  Set to 1.0 for linear gravity
- physical_parameters/**g0** *[real]*
  Set to 1.0 for constant gravity
- physical_parameters/**radratio** *[real]*
  aspect ratio
- physical_parameters/**con_funcwidth** *[real]*
  nVarCond=1

- physical_parameters/**kbotb** *[integer]*
- physical_parameters/**nvareps** *[integer]*
  Selection of internal heating profile
- physical_parameters/**con_radratio** *[real]*
  Transition between branches of electrical conductivity profile (nVarCond=1,2)
- physical_parameters/**slopestrat** *[real]*
  stratified Layer
- physical_parameters/**nvarvisc** *[integer]*
  Selection of variable viscosity profile
- physical_parameters/**n_imps_max** *[integer,parameter*=20*]*
  Heat boundary condition
- physical_parameters/**imps** *[integer]*
  Heat boundary condition
- physical_parameters/**prmag** *[real]*
  magnetic Prandtl number
- physical_parameters/**r_lcr** *[real]*
  Radius beyond which conductivity is zero
- physical_parameters/**difexp** *[real]*
  Thermal diffusivity variation
- physical_parameters/**nvardiff** *[integer]*
  Selection of variable diffusivity profile
- physical_parameters/**con_lambdaout** *[real]*
  nVarCond=1
- physical_parameters/**opr** *[real]*
  Inverse of Prandtl number
- physical_parameters/**kbotv** *[integer]*
- physical_parameters/**epss** *[real]*
  Deviation from the adiabat
- physical_parameters/**phis** (20) *[real]*
- physical_parameters/**imagcon** *[integer]*
  Imposed magnetic field for magnetoconvection, at the boundaries
- physical_parameters/**widths** (20) *[real]*
- physical_parameters/**interior_model** *[character]*
  name of the interior model
- physical_parameters/**nvarcond** *[integer]*
  Selection of variable conductivity profile
- physical_parameters/**ekscaled** *[real]*
  $E\,l^2$
- physical_parameters/**rascaled** *[real]*
  $Ra\,l^3$
- physical_parameters/**tmagcon** *[real]*
  Time for magnetoconvection calculation

10.2.5 logic.f90

Description

Module containing the logicals that control the run

Quick access

Variables *l_pressgraph*, *l_cmb_field*, *l_dt_cmb_field*, *l_corrmov*, *l_rms*, *l_tomovie*, *l_to*, *l_runtimelimit*, *l_par*, *l_anel*, *l_anelastic_liquid*, *l_am*, *l_perppar*, *l_fluxprofs*, *l_average*, *l_cond_ic*, *l_cond_ma*, *l_correct_amz*, *l_correct_ame*, *lverbose*, *l_viscbccalc*, *l_drift*, *l_dtbmovie*, *l_ht*, *l_htmovie*, *l_hel*, *l_iner*, *l_isothermal*, *l_rmagspec*, *l_dtrmagspec*, *l_b_nl_icb*, *l_b_nl_cmb*, *l_newmap*, *l_pv*, *l_power*, *l_r_fieldt*, *l_r_field*, *l_movie*, *l_movie_ic*, *l_movie_oc*, *l_z10mat*, *l_storepot*, *l_storetpot*, *l_storebpot*, *l_store_frame*, *l_storevpot*, *l_time_hits*, *l_true_time*, *l_lcr*, *l_rot_ic*, *l_mag_lf*, *l_rot_ma*, *l_conv*, *l_update_s*, *l_heat*, *l_update_b*, *l_mag*, *l_conv_nl*, *l_heat_nl*, *l_mag_nl*, *l_save_out*, *l_corr*, *l_update_v*, *l_mag_kin*, *l_non_rot*, *l_dtb*, *l_sric*, *l_srma*

Variables

- logic/**l_time_hits** *[logical]*
  Switch for time for outputs
- logic/**l_to** *[logical]*
  Switch for TO output in TOnhs.TAG, TOshs.TAG
- logic/**l_dtrmagspec** *[logical]*
  Switch for magnetic spectra at different depths at movie output times
- logic/**l_lcr** *[logical]*
  Switch for zero electrical conductivity beyond r_LCR
- logic/**l_dtbmovie** *[logical]*
  Switch for dtB movie
- logic/**lverbose** *[logical]*
  Switch for detailed information about run progress
- logic/**l_correct_amz** *[logical]*
  Switch for correction of axial angular momentum
- logic/**l_correct_ame** *[logical]*
  Switch for correction of equatorial angular mom.
- logic/**l_pv** *[logical]*
  Switch for potential vorticity calculation
- logic/**l_heat** *[logical]*
  Switch off heat terms calculation
- logic/**l_srma** *[logical]*
  Switch to rotating OC with prescribed rot. rate
- logic/**l_mag_lf** *[logical]*
  Switch off Lorentz force term

- logic/**l_perppar** *[logical]*
  Switch for calculation of of kinetic energy perpendicular+parallel to the rotation axis
- logic/**l_sric** *[logical]*
  Switch to rotating IC with prescribed rot. rate
- logic/**l_mag** *[logical]*
  Switch off magnetic terms calculation
- logic/**l_non_rot** *[logical]*
  Switch to non-rotating
- logic/**l_update_b** *[logical]*
  Switch off magnetic field update
- logic/**l_r_fieldt** *[logical]*
  Switch for radial T coefficients
- logic/**l_average** *[logical]*
  Switch for calculation of time-averages
- logic/**l_update_v** *[logical]*
  Switch off velocity field update
- logic/**l_cond_ic** *[logical]*
  Switch for conducting IC
- logic/**l_update_s** *[logical]*
  Switch off entropy update
- logic/**l_storevpot** *[logical]*
  Switch for storing velocity field potentials
- logic/**l_am** *[logical]*
  Switch for angular momentum calculation
- logic/**l_storebpot** *[logical]*
  Switch for storing magnetic field potentials
- logic/**l_viscbccalc** *[logical]*
  Switch for dissipation layer for stress-free BCs plots
- logic/**l_conv** *[logical]*
  Switch off convection
- logic/**l_pressgraph** *[logical]*
  Store pressure in graphic files
- logic/**l_true_time** *[logical]*
  Switch for times of outputs
- logic/**l_movie_ic** *[logical]*
  Switch for recording of movie files for IC
- logic/**l_store_frame** *[logical]*
  Switch for storing movie frames
- logic/**l_isothermal** *[logical]*
  Switch for isothermal calculation
- logic/**l_mag_nl** *[logical]*
  Switch off non-linear magnetic terms calculation

- logic/**l_z10mat** *[logical]*
  Switch for solid body rotation
- logic/**l_rmagspec** *[logical]*
  Switch for magnetic spectra at different depths at log times
- logic/**l_drift** *[logical]*
  Switch for drift rates calculation
- logic/**l_dt_cmb_field** *[logical]*
  Switch for Bcoef files for secular variation of gauss coefs.
- logic/**l_hel** *[logical]*
  Switch for helicity calculation, output in misc.TAG
- logic/**l_corrmov** *[logical]*
  Switch for North/south correlation movie (see s_getEgeos.f)
- logic/**l_newmap** *[logical]*
  Switch for non-linear mapping (see Bayliss and Turkel, 1990)
- logic/**l_storetpot** *[logical]*
  Switch for storing entropy field potentials
- logic/**l_cond_ma** *[logical]*
  Switch for conducting OC
- logic/**l_save_out** *[logical]*
  Switch off outputs
- logic/**l_rot_ic** *[logical]*
  Switch off IC rotation
- logic/**l_anelastic_liquid** *[logical]*
  Switch for anelastic liquid calculation
- logic/**l_b_nl_icb** *[logical]*
  Switch for non-linear magnetic field at IC
- logic/**l_conv_nl** *[logical]*
  Switch off non-linear convection terms
- logic/**l_htmovie** *[logical]*
  Switch for heat flux movie output
- logic/**l_movie** *[logical]*
  Switch for recording of movie files
- logic/**l_anel** *[logical]*
  Switch for anelastic calculation
- logic/**l_runtimelimit** *[logical]*
  Switch for absolute time limit of the run
- logic/**l_fluxprofs** *[logical]*
  Switch for calculation of radial profiles of flux contributions
- logic/**l_movie_oc** *[logical]*
  Switch for recording of movie files for OC
- logic/**l_cmb_field** *[logical]*
  Switch for Bcoef files for gauss coefficients

- `logic/`**`l_mag_kin`** *[logical]*
  Switch related for kinematic dynamo
- `logic/`**`l_heat_nl`** *[logical]*
  Switch off non-linear heat terms calculation
- `logic/`**`l_iner`** *[logical]*
  Switch for inertial modes calculation
- `logic/`**`l_rot_ma`** *[logical]*
  Switch off OC rotation
- `logic/`**`l_ht`** *[logical]*
  Switch for heat flux movie frame output
- `logic/`**`l_storepot`** *[logical]*
  Switch for storing all field potentials
- `logic/`**`l_tomovie`** *[logical]*
  Switch for TO movie output
- `logic/`**`l_power`** *[logical]*
  Switch for power budget terms calculation
- `logic/`**`l_dtb`** *[logical]*
  Switch to reserve memory for dtB movie
- `logic/`**`l_par`** *[logical]*
  Switch for additional parameters calculation in s_getEgeos.f
- `logic/`**`l_b_nl_cmb`** *[logical]*
  Switch for non-linear magnetic field at OC
- `logic/`**`l_rms`** *[logical]*
  Switch for RMS force balances calculation
- `logic/`**`l_r_field`** *[logical]*
  Switch for radial coefficients
- `logic/`**`l_corr`** *[logical]*
  Switch off rotation

10.2.6 `fields.f90`

Description

This module contains the potential fields and their radial derivatives

Quick access

Variables *omega_ic*, *omega_ma*, *aj*, *aj_ic*, *aj_ic_lmloc*, *b*, *b_ic*, *b_ic_lmloc*, *db*, *db_ic*, *db_ic_lmloc*, *ddb*, *ddb_ic*, *ddb_ic_lmloc*, *ddj*, *ddj_ic*, *ddj_ic_lmloc*, *ddw*, *dj*, *dj_ic*, *dj_ic_lmloc*, *dp*, *ds*, *dw*, *dz*, *p*, *s*, *w*, *z*, *aj_lmloc*, *aj_rloc*, *b_lmloc*, *b_rloc*, *db_lmloc*, *db_rloc*, *ddb_lmloc*, *ddb_rloc*, *ddj_lmloc*, *ddw_lmloc*, *ddw_rloc*, *dj_lmloc*, *dj_rloc*, *dp_lmloc*, *dp_rloc*, *ds_lmloc*, *ds_rloc*, *dw_lmloc*, *dw_rloc*, *dz_lmloc*, *dz_rloc*, *p_lmloc*, *p_rloc*, *s_lmloc*, *s_rloc*, *w_lmloc*, *w_rloc*, *z_lmloc*, *z_rloc*, *aj_lmloc_container*, *aj_rloc_container*, *b_lmloc_container*,

b_rloc_container, *p_lmloc_container*, *p_rloc_container*, *s_lmloc_container*, *s_rloc_container*, *w_lmloc_container*, *w_rloc_container*, *z_lmloc_container*, *z_rloc_container*

Routines *initialize_fields()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod* (`rank()`): This module contains the blocking information
- *radial_data* (*nrstart()*, *nrstop()*)
- *lmloop_data* (*ulmmag()*, *llmmag()*, *llm()*, *ulm()*)
- *truncation* (*lm_max()*, *lm_maxmag()*, *n_r_ic_maxmag()*, *n_r_max()*, *n_r_maxmag()*): This module defines the grid points and the truncation

Variables

- `fields/`**`aj_lmloc`** (:,:) *[complex,pointer/public]*
- `fields/`**`ddj_ic`** (:,:) *[complex,allocatable/public]*
- `fields/`**`ddw_rloc`** (:,:) *[complex,pointer/public]*
- `fields/`**`ddw_lmloc`** (:,:) *[complex,pointer/public]*
- `fields/`**`b_lmloc`** (:,:) *[complex,pointer/public]*
- `fields/`**`dj_lmloc`** (:,:) *[complex,pointer/public]*
- `fields/`**`p_rloc`** (:,:) *[complex,pointer/public]*
- `fields/`**`aj_lmloc_container`** (:,:,:) *[complex,target/allocatable/public]*
- `fields/`**`db_lmloc`** (:,:) *[complex,pointer/public]*
- `fields/`**`s_lmloc`** (:,:) *[complex,pointer/public]*
- `fields/`**`b_ic`** (:,:) *[complex,allocatable/public]*
- `fields/`**`z_rloc`** (:,:) *[complex,pointer/public]*
- `fields/`**`dj_rloc`** (:,:) *[complex,pointer/public]*
- `fields/`**`p_lmloc`** (:,:) *[complex,pointer/public]*
- `fields/`**`dw_lmloc`** (:,:) *[complex,pointer/public]*
- `fields/`**`dp_lmloc`** (:,:) *[complex,pointer/public]*
- `fields/`**`dj_ic_lmloc`** (:,:) *[complex,allocatable/public]*
- `fields/`**`ddb_ic`** (:,:) *[complex,allocatable/public]*
- `fields/`**`dj`** (:,:) *[complex,allocatable/public]*
- `fields/`**`s_lmloc_container`** (:,:,:) *[complex,target/allocatable/public]*
- `fields/`**`db`** (:,:) *[complex,allocatable/public]*
- `fields/`**`aj_ic_lmloc`** (:,:) *[complex,allocatable/public]*
- `fields/`**`dz`** (:,:) *[complex,allocatable/public]*

- fields/**z_lmloc** (:,:) *[complex,pointer/public]*
- fields/**dz_rloc** (:,:) *[complex,pointer/public]*
- fields/**dw** (:,:) *[complex,allocatable/public]*
- fields/**w_lmloc** (:,:) *[complex,pointer/public]*
- fields/**ds** (:,:) *[complex,allocatable/public]*
- fields/**dp** (:,:) *[complex,allocatable/public]*
- fields/**b** (:,:) *[complex,allocatable/public]*
- fields/**ddj_ic_lmloc** (:,:) *[complex,allocatable/public]*
- fields/**ddb_rloc** (:,:) *[complex,pointer/public]*
- fields/**db_rloc** (:,:) *[complex,pointer/public]*
- fields/**w_rloc** (:,:) *[complex,pointer/public]*
- fields/**p** (:,:) *[complex,allocatable/public]*
- fields/**w_rloc_container** (:,:,:) *[complex,target/allocatable/public]*
- fields/**z** (:,:) *[complex,allocatable/public]*
- fields/**s_rloc_container** (:,:,:) *[complex,target/allocatable/public]*
- fields/**aj_rloc_container** (:,:,:) *[complex,target/allocatable/public]*
- fields/**aj_ic** (:,:) *[complex,allocatable/public]*
- fields/**aj** (:,:) *[complex,allocatable/public]*
- fields/**dw_rloc** (:,:) *[complex,pointer/public]*
- fields/**dp_rloc** (:,:) *[complex,pointer/public]*
- fields/**s_rloc** (:,:) *[complex,pointer/public]*
- fields/**ddb_lmloc** (:,:) *[complex,pointer/public]*
- fields/**w_lmloc_container** (:,:,:) *[complex,target/allocatable/public]*
- fields/**ds_lmloc** (:,:) *[complex,pointer/public]*
- fields/**b_rloc_container** (:,:,:) *[complex,target/allocatable/public]*
- fields/**ddb_ic_lmloc** (:,:) *[complex,allocatable/public]*
- fields/**ddw** (:,:) *[complex,allocatable/public]*
- fields/**ddj** (:,:) *[complex,allocatable/public]*
- fields/**ddb** (:,:) *[complex,allocatable/public]*
- fields/**db_ic_lmloc** (:,:) *[complex,allocatable/public]*
- fields/**db_ic** (:,:) *[complex,allocatable/public]*
- fields/**dj_ic** (:,:) *[complex,allocatable/public]*
- fields/**p_rloc_container** (:,:,:) *[complex,target/allocatable/public]*
- fields/**z_lmloc_container** (:,:,:) *[complex,target/allocatable/public]*
- fields/**omega_ic** *[real,public]*
- fields/**p_lmloc_container** (:,:,:) *[complex,target/allocatable/public]*

- fields/**ds_rloc** (:,:) *[complex,pointer/public]*
- fields/**omega_ma** *[real,public]*
- fields/**b_lmloc_container** (:,:,:) *[complex,target/allocatable/public]*
- fields/**aj_rloc** (:,:) *[complex,pointer/public]*
- fields/**b_ic_lmloc** (:,:) *[complex,allocatable/public]*
- fields/**b_rloc** (:,:) *[complex,pointer/public]*
- fields/**z_rloc_container** (:,:,:) *[complex,target/allocatable/public]*
- fields/**s** (:,:) *[complex,allocatable/public]*
- fields/**ddj_lmloc** (:,:) *[complex,pointer/public]*
- fields/**w** (:,:) *[complex,allocatable/public]*
- fields/**dz_lmloc** (:,:) *[complex,pointer/public]*

Subroutines and functions

subroutine fields/**initialize_fields**()
Memory allocation

Called from *magic*

10.2.7 dt_fieldsLast.f90

Description

This module contains time-derivaties array of the previous time-step They are needed in the time-stepping scheme.

The variables labeled with a suffix 'Last' are provided by the restart file for the first time step or calculated here or by the update routines for the following time step. These fields remain in the LM-distributed space

Quick access

Variables *d_omega_ic_dtlast*, *d_omega_ma_dtlast*, *lorentz_torque_iclast*, *lorentz_torque_malast*, *dbdt_iclast*, *dbdt_iclast_lmloc*, *dbdtlast*, *dbdtlast_lmloc*, *djdt_iclast*, *djdt_iclast_lmloc*, *djdtlast*, *djdtlast_lmloc*, *dpdtlast*, *dpdtlast_lmloc*, *dsdtlast*, *dsdtlast_lmloc*, *dwdtlast*, *dwdtlast_lmloc*, *dzdtlast*, *dzdtlast_lo*

Routines *initialize_fieldslast()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod* (rank()): This module contains the blocking information
- *lmloop_data* (*ulmmag()*, *llmmag()*, *llm()*, *ulm()*)
- *truncation* (*lm_max()*, *lm_maxmag()*, *n_r_ic_maxmag()*, *n_r_max()*, *n_r_maxmag()*): This module defines the grid points and the truncation

Variables

- fieldslast/**dbdt_iclast_lmloc** (:,:) *[complex,allocatable/public]*
- fieldslast/**djdtlast_lmloc** (:,:) *[complex,allocatable/public]*
- fieldslast/**dpdtlast** (:,:) *[complex,allocatable/public]*
- fieldslast/**dpdtlast_lmloc** (:,:) *[complex,allocatable/public]*
- fieldslast/**lorentz_torque_malast** *[real,public]*
- fieldslast/**djdt_iclast** (:,:) *[complex,allocatable/public]*
- fieldslast/**dsdtlast** (:,:) *[complex,allocatable/public]*
- fieldslast/**dwdtlast_lmloc** (:,:) *[complex,allocatable/public]*
- fieldslast/**dzdtlast** (:,:) *[complex,allocatable/public]*
- fieldslast/**dzdtlast_lo** (:,:) *[complex,allocatable/public]*
- fieldslast/**lorentz_torque_iclast** *[real,public]*
- fieldslast/**djdt_iclast_lmloc** (:,:) *[complex,allocatable/public]*
- fieldslast/**d_omega_ma_dtlast** *[real,public]*
- fieldslast/**djdtlast** (:,:) *[complex,allocatable/public]*
- fieldslast/**d_omega_ic_dtlast** *[real,public]*
- fieldslast/**dbdtlast** (:,:) *[complex,allocatable/public]*
- fieldslast/**dsdtlast_lmloc** (:,:) *[complex,allocatable/public]*
- fieldslast/**dbdtlast_lmloc** (:,:) *[complex,allocatable/public]*
- fieldslast/**dbdt_iclast** (:,:) *[complex,allocatable/public]*
- fieldslast/**dwdtlast** (:,:) *[complex,allocatable/public]*

Subroutines and functions

subroutine fieldslast/**initialize_fieldslast**()
Memory allocation

Called from *magic*

10.2.8 mat.f90

Description

This module contains matricies for internal time step

Quick access

Variables *lz10mat*, *lbmat*, *lsmat*, *lwpmat*, *lzmat*, *p0pivot*, *s0pivot*, *z10pivot*, *bpivot*, *jpivot*, *spivot*, *wppivot*, *zpivot*, *s0mat_fac*, *z10mat_fac*, *bmat_fac*, *jmat_fac*, *p0mat*, *s0mat*, *smat_fac*, *z10mat*, *zmat_fac*, *bmat*, *jmat*, *smat*, *wpmat*, *wpmat_fac*, *zmat*

Routines *initialize_matrices()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *truncation* (*n_r_totmag()*, *l_max()*, *l_maxmag()*, *n_r_max()*, *n_r_tot()*): This module defines the grid points and the truncation

Variables

- matrices/**jpivot** (:,:) *[integer,allocatable/public]*
- matrices/**lbmat** (:) *[logical,allocatable/public]*
- matrices/**smat** (:,:,:) *[real,allocatable/public]*
- matrices/**zmat** (:,:,:) *[real,allocatable/public]*
- matrices/**p0pivot** (:) *[integer,allocatable/public]*
- matrices/**smat_fac** (:,:) *[real,allocatable/public]*
- matrices/**lz10mat** *[logical,public]*
- matrices/**z10mat_fac** (:) *[real,allocatable/public]*
- matrices/**jmat_fac** (:,:) *[real,allocatable/public]*
- matrices/**s0mat** (:,:) *[real,allocatable/public]*
- matrices/**lsmat** (:) *[logical,allocatable/public]*
- matrices/**lwpmat** (:) *[logical,allocatable/public]*
- matrices/**s0pivot** (:) *[integer,allocatable/public]*
- matrices/**lzmat** (:) *[logical,allocatable/public]*
- matrices/**wpmat_fac** (:,:,:) *[real,allocatable/public]*
- matrices/**wppivot** (:,:) *[integer,allocatable/public]*
- matrices/**p0mat** (:,:) *[real,allocatable/public]*
- matrices/**zmat_fac** (:,:) *[real,allocatable/public]*
- matrices/**bpivot** (:,:) *[integer,allocatable/public]*
- matrices/**bmat_fac** (:,:) *[real,allocatable/public]*
- matrices/**bmat** (:,:,:) *[real,allocatable/public]*
- matrices/**wpmat** (:,:,:) *[real,allocatable/public]*
- matrices/**z10pivot** (:) *[integer,allocatable/public]*
- matrices/**jmat** (:,:,:) *[real,allocatable/public]*
- matrices/**zpivot** (:,:) *[integer,allocatable/public]*
- matrices/**s0mat_fac** (:) *[real,allocatable/public]*
- matrices/**z10mat** (:,:) *[real,allocatable/public]*
- matrices/**spivot** (:,:) *[integer,allocatable/public]*

Subroutines and functions

subroutine matrices/**initialize_matrices**()

Called from *magic*

10.2.9 output_data.f90

Description

This module contains the parameters for output control

Quick access

Variables *graph_mpi_fh*, *l_graph_time*, *l_max_cmb*, *l_max_r*, *n_angular_file*, *n_bpot_step*, *n_bpots*, *n_cmb_file*, *n_cmb_step*, *n_cmbmov_file*, *n_cmbs*, *n_coeff_r_max*, *n_dipole_file*, *n_dt_cmb_file*, *n_dtbrms_file*, *n_dtvrms_file*, *n_e_kin_file*, *n_e_mag_ic_file*, *n_e_mag_oc_file*, *n_graph_file*, *n_graph_step*, *n_graphs*, *n_kin_spec_file*, *n_log_file*, *n_log_step*, *n_logs*, *n_lp_file*, *n_mag_spec_file*, *n_misc_file*, *n_movie_frames*, *n_movie_step*, *n_par_file*, *n_perppar_file*, *n_pot_step*, *n_pots*, *n_power_file*, *n_r_field_step*, *n_r_fields*, *n_r_step*, *n_rot_file*, *n_rst_file*, *n_rst_step*, *n_rsts*, *n_signal_file*, *n_spec_step*, *n_specs*, *n_sric_file*, *n_srma_file*, *n_stores*, *n_t_bpot*, *n_t_cmb*, *n_t_graph*, *n_t_log*, *n_t_movie*, *n_t_pot*, *n_t_r_field*, *n_t_rst*, *n_t_spec*, *n_t_to*, *n_t_tomovie*, *n_t_toz*, *n_t_tpot*, *n_t_vpot*, *n_to_step*, *n_tomovie_frames*, *n_tomovie_step*, *n_tos*, *n_toz_step*, *n_tozs*, *n_tpot_step*, *n_tpots*, *n_u2_spec_file*, *n_u_square_file*, *n_vpot_step*, *n_vpots*, *nlf*, *rst_mpi_fh*, *n_time_hits*, *nsmaxa*, *n_b_r_file*, *n_coeff_r*, *n_t_r_file*, *n_v_r_file*, *tag*, *runid*, *angular_file*, *cmb_file*, *cmbmov_file*, *dipole_file*, *dt_cmb_file*, *dtbrms_file*, *dtvrms_file*, *e_kin_file*, *e_mag_ic_file*, *e_mag_oc_file*, *graph_file*, *log_file*, *lp_file*, *misc_file*, *par_file*, *perppar_file*, *power_file*, *rot_file*, *rst_file*, *sric_file*, *srma_file*, *u_square_file*, *n_r_array*, *dt_bpot*, *dt_cmb*, *dt_graph*, *dt_log*, *dt_movie*, *dt_pot*, *dt_r_field*, *dt_rst*, *dt_spec*, *dt_to*, *dt_tomovie*, *dt_toz*, *dt_tpot*, *dt_vpot*, *rcut*, *rdea*, *sdens*, *t_bpot_start*, *t_bpot_stop*, *t_cmb_start*, *t_cmb_stop*, *t_graph_start*, *t_graph_stop*, *t_log_start*, *t_log_stop*, *t_movie_start*, *t_movie_stop*, *t_pot_start*, *t_pot_stop*, *t_r_field_start*, *t_r_field_stop*, *t_rst_start*, *t_rst_stop*, *t_spec_start*, *t_spec_stop*, *t_to_start*, *t_to_stop*, *t_tomovie_start*, *t_tomovie_stop*, *t_toz_start*, *t_toz_stop*, *t_tpot_start*, *t_tpot_stop*, *t_vpot_start*, *t_vpot_stop*, *zdens*, *nzmaxa*, *b_r_file*, *t_r_file*, *v_r_file*, *t_bpot*, *t_cmb*, *t_graph*, *t_log*, *t_movie*, *t_pot*, *t_r_field*, *t_rst*, *t_spec*, *t_to*, *t_tomovie*, *t_toz*, *t_tpot*, *t_vpot*

Routines *closefiles()*, *openfiles()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod* (rank()): This module contains the blocking information

- *charmanip* (*length_to_blank()*): This module contains several useful routines to manipule character strings
- *logic* (*l_rot_ic()*, *l_sric()*, *l_dt_cmb_field()*, *l_power()*, *l_r_fieldt()*, *l_cmb_field()*, *l_rot_ma()*, *l_save_out()*, *l_srma()*, *l_movie()*, *l_anel()*, *l_perppar()*, *l_am()*, *l_rms()*, *l_r_field()*, *l_mag()*): Module containing the logicals that control the run

Variables

- output_data/**n_tozs** *[integer,public]*
- output_data/**n_graphs** *[integer,public]*
- output_data/**t_toz_start** *[real,public]*
- output_data/**rst_file** *[character,public]*
- output_data/**t_spec** (5000) *[real,public]*
- output_data/**n_tpots** *[integer,public]*
- output_data/**runid** *[character,public]*
- output_data/**nlf** *[integer,public]*
- output_data/**n_movie_frames** *[integer,public]*
- output_data/**n_rst_file** *[integer,public]*
- output_data/**dt_spec** *[real,public]*
- output_data/**dt_tpot** *[real,public]*
- output_data/**power_file** *[character,public]*
- output_data/**n_r_step** *[integer,public]*
- output_data/**n_par_file** *[integer,public]*
- output_data/**dt_cmb_file** *[character,public]*
- output_data/**t_r_file** (:) *[character,allocatable/public]*
- output_data/**t_movie_start** *[real,public]*
- output_data/**n_t_toz** *[integer,public]*
- output_data/**n_t_cmb** *[integer,public]*
- output_data/**sdens** *[real,public]*
- output_data/**n_tpot_step** *[integer,public]*
- output_data/**n_t_spec** *[integer,public]*
- output_data/**t_rst_start** *[real,public]*
- output_data/**n_time_hits** *[integer,parameter*=5000/public*]*
- output_data/**t_bpot_start** *[real,public]*
- output_data/**dt_graph** *[real,public]*
- output_data/**dt_tomovie** *[real,public]*
- output_data/**t_rst_stop** *[real,public]*

- output_data/**t_vpot_start** *[real,public]*
- output_data/**n_r_array** (100) *[integer,public]*
- output_data/**rst_mpi_fh** *[integer,public]*
- output_data/**n_t_tomovie** *[integer,public]*
- output_data/**l_max_cmb** *[integer,public]*
- output_data/**n_to_step** *[integer,public]*
- output_data/**n_u_square_file** *[integer,public]*
- output_data/**n_t_log** *[integer,public]*
- output_data/**n_log_step** *[integer,public]*
- output_data/**dt_movie** *[real,public]*
- output_data/**par_file** *[character,public]*
- output_data/**t_rst** (5000) *[real,public]*
- output_data/**t_log_stop** *[real,public]*
- output_data/**dt_cmb** *[real,public]*
- output_data/**v_r_file** (:) *[character,allocatable/public]*
- output_data/**n_power_file** *[integer,public]*
- output_data/**n_cmb_step** *[integer,public]*
- output_data/**l_graph_time** *[logical,public]*
- output_data/**dt_r_field** *[real,public]*
- output_data/**n_dt_cmb_file** *[integer,public]*
- output_data/**nzmaxa** *[integer,parameter*=194/public*]*
- output_data/**t_graph_start** *[real,public]*
- output_data/**dt_vpot** *[real,public]*
- output_data/**n_t_r_file** (:) *[integer,allocatable/public]*
- output_data/**e_mag_ic_file** *[character,public]*
- output_data/**dt_log** *[real,public]*
- output_data/**t_cmb** (5000) *[real,public]*
- output_data/**n_t_pot** *[integer,public]*
- output_data/**n_graph_step** *[integer,public]*
- output_data/**n_t_bpot** *[integer,public]*
- output_data/**zdens** *[real,public]*
- output_data/**t_graph** (5000) *[real,public]*
- output_data/**t_to_start** *[real,public]*
- output_data/**dt_toz** *[real,public]*
- output_data/**n_dtvrms_file** *[integer,public]*
- output_data/**n_bpots** *[integer,public]*

- output_data/**perppar_file** *[character,public]*
- output_data/**n_misc_file** *[integer,public]*
- output_data/**graph_mpi_fh** *[integer,public]*
- output_data/**t_r_field_stop** *[real,public]*
- output_data/**n_log_file** *[integer,public]*
- output_data/**cmbmov_file** *[character,public]*
- output_data/**n_rot_file** *[integer,public]*
- output_data/**n_specs** *[integer,public]*
- output_data/**t_cmb_start** *[real,public]*
- output_data/**t_tomovie_stop** *[real,public]*
- output_data/**t_r_field_start** *[real,public]*
- output_data/**n_tos** *[integer,public]*
- output_data/**n_rst_step** *[integer,public]*
- output_data/**dt_pot** *[real,public]*
- output_data/**b_r_file** (:) *[character,allocatable/public]*
- output_data/**t_vpot** (5000) *[real,public]*
- output_data/**n_graph_file** *[integer,public]*
- output_data/**n_perppar_file** *[integer,public]*
- output_data/**n_vpot_step** *[integer,public]*
- output_data/**dtvrms_file** *[character,public]*
- output_data/**t_tpot_start** *[real,public]*
- output_data/**n_sric_file** *[integer,public]*
- output_data/**n_t_movie** *[integer,public]*
- output_data/**n_t_to** *[integer,public]*
- output_data/**n_dtbrms_file** *[integer,public]*
- output_data/**rot_file** *[character,public]*
- output_data/**n_tomovie_step** *[integer,public]*
- output_data/**t_cmb_stop** *[real,public]*
- output_data/**n_vpots** *[integer,public]*
- output_data/**n_rsts** *[integer,public]*
- output_data/**t_log_start** *[real,public]*
- output_data/**t_toz_stop** *[real,public]*
- output_data/**t_to** (5000) *[real,public]*
- output_data/**n_tomovie_frames** *[integer,public]*
- output_data/**n_pots** *[integer,public]*
- output_data/**n_kin_spec_file** *[integer,public]*

- output_data/**t_tpot** (5000) *[real,public]*
- output_data/**n_cmbmov_file** *[integer,public]*
- output_data/**n_u2_spec_file** *[integer,public]*
- output_data/**n_cmbs** *[integer,public]*
- output_data/**n_t_graph** *[integer,public]*
- output_data/**dt_rst** *[real,public]*
- output_data/**n_srma_file** *[integer,public]*
- output_data/**t_tomovie_start** *[real,public]*
- output_data/**t_toz** (5000) *[real,public]*
- output_data/**dt_to** *[real,public]*
- output_data/**dtbrms_file** *[character,public]*
- output_data/**n_e_mag_oc_file** *[integer,public]*
- output_data/**n_v_r_file** (:) *[integer,allocatable/public]*
- output_data/**n_stores** *[integer,public]*
- output_data/**n_t_vpot** *[integer,public]*
- output_data/**t_pot_start** *[real,public]*
- output_data/**sric_file** *[character,public]*
- output_data/**e_mag_oc_file** *[character,public]*
- output_data/**n_coeff_r** (:) *[integer,allocatable/public]*
- output_data/**e_kin_file** *[character,public]*
- output_data/**misc_file** *[character,public]*
- output_data/**n_logs** *[integer,public]*
- output_data/**t_movie** (5000) *[real,public]*
- output_data/**n_t_tpot** *[integer,public]*
- output_data/**t_tpot_stop** *[real,public]*
- output_data/**angular_file** *[character,public]*
- output_data/**n_signal_file** *[integer,public]*
- output_data/**l_max_r** *[integer,public]*
- output_data/**n_r_fields** *[integer,public]*
- output_data/**n_movie_step** *[integer,public]*
- output_data/**t_r_field** (5000) *[real,public]*
- output_data/**n_t_r_field** *[integer,public]*
- output_data/**n_pot_step** *[integer,public]*
- output_data/**t_pot** (5000) *[real,public]*
- output_data/**t_graph_stop** *[real,public]*
- output_data/**nsmaxa** *[integer,parameter*=97/public*]*

- output_data/**n_e_kin_file** *[integer,public]*
- output_data/**n_r_field_step** *[integer,public]*
- output_data/**rcut** *[real,public]*
- output_data/**t_movie_stop** *[real,public]*
- output_data/**t_log** (5000) *[real,public]*
- output_data/**tag** *[character,public]*
- output_data/**n_toz_step** *[integer,public]*
- output_data/**n_coeff_r_max** *[integer,public]*
- output_data/**dipole_file** *[character,public]*
- output_data/**n_bpot_step** *[integer,public]*
- output_data/**lp_file** *[character,public]*
- output_data/**u_square_file** *[character,public]*
- output_data/**t_spec_start** *[real,public]*
- output_data/**n_angular_file** *[integer,public]*
- output_data/**n_t_rst** *[integer,public]*
- output_data/**cmb_file** *[character,public]*
- output_data/**graph_file** *[character,public]*
- output_data/**n_spec_step** *[integer,public]*
- output_data/**t_bpot_stop** *[real,public]*
- output_data/**t_spec_stop** *[real,public]*
- output_data/**t_bpot** (5000) *[real,public]*
- output_data/**n_e_mag_ic_file** *[integer,public]*
- output_data/**srma_file** *[character,public]*
- output_data/**t_tomovie** (5000) *[real,public]*
- output_data/**dt_bpot** *[real,public]*
- output_data/**n_cmb_file** *[integer,public]*
- output_data/**n_lp_file** *[integer,public]*
- output_data/**t_pot_stop** *[real,public]*
- output_data/**t_to_stop** *[real,public]*
- output_data/**t_vpot_stop** *[real,public]*
- output_data/**n_b_r_file** (:) *[integer,allocatable/public]*
- output_data/**n_dipole_file** *[integer,public]*
- output_data/**n_mag_spec_file** *[integer,public]*
- output_data/**log_file** *[character,public]*
- output_data/**rdea** *[real,public]*

Subroutines and functions

subroutine output_data/**openfiles**()
Defines names and unit for output files and opens then.

Called from *magic*

Call to *length_to_blank()*

subroutine output_data/**closefiles**()
Defines names and unit for output files and opens then. MPI: called only by the processor responsible for output.

Called from *magic*

10.2.10 Bext.f90

Description

Module containing the external field parameters

Quick access

Variables *n_imp*, *l_imp*, *l_curr*, *amp_curr*, *amp_imp*, *expo_imp*, *bmax_imp*, *rrmp*, *fac_loop*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC

Variables

- bext/**bmax_imp** *[real]*
  Location of maximum in g_ext/g_int
- bext/**l_curr** *[logical]*
  Switch for current loop at the equator
- bext/**amp_curr** *[real]*
  Amplitude of magnetic field of current loop
- bext/**fac_loop** (:) *[real,allocatable]*
- bext/**rrmp** *[real]*
  Magnetopause radius
- bext/**n_imp** *[integer]*
  Controls external field model
- bext/**l_imp** *[integer]*
  Mode of external field (dipole,quadrupole etc.)
- bext/**expo_imp** *[real]*
  Exponent for decay
- bext/**amp_imp** *[real]*
  Amplitude of the time varying osc

10.2.11 constants.f90

Description

module containing constants and parameters used in the code.

Quick access

Variables *c_dt_z10_ic*, *c_dt_z10_ma*, *c_lorentz_ic*, *c_lorentz_ma*, *c_z10_omega_ic*, *c_z10_omega_ma*, *y10_norm*, *y11_norm*, *mass*, *c_moi_ic*, *c_moi_ma*, *c_moi_oc*, *surf_cmb*, *vol_ic*, *vol_oc*, *codeversion*, *zero*, *ci*, *half*, *sin60*, *one*, *two*, *three*, *four*, *pi*, *osq4pi*, *cos36*, *cos72*, *third*, *sin36*, *sin72*, *sq4pi*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC

Variables

- constants/**ci** *[complex,parameter*=(0.0_cp,*1.0_cp)]*
- constants/**c_z10_omega_ma** *[real]*
- constants/**c_lorentz_ic** *[real]*
- constants/**one** *[real,parameter*=1.0_cp*]*
- constants/**four** *[real,parameter*=4.0_cp*]*
- constants/**sin60** *[real,parameter*=0.5_cp*sqrt(3.0_cp)*]*
- constants/**zero** *[complex,parameter*=(0.0_cp,*0.0_cp)]*
- constants/**surf_cmb** *[real]*
  Outer boundary surface
- constants/**c_moi_oc** *[real]*
  Moment of inertia of the outer core
- constants/**three** *[real,parameter*=3.0_cp*]*
- constants/**c_dt_z10_ic** *[real]*
- constants/**c_z10_omega_ic** *[real]*
- constants/**c_moi_ic** *[real]*
  Moment of inertia of the inner core
- constants/**c_moi_ma** *[real]*
  Moment of inertia of the mantle
- constants/**pi** *[real,parameter*=4.0_cp*atan(1.0_cp)*]*
- constants/**osq4pi** *[real,parameter*=1.0_cp/*sq4pi*]*
- constants/**cos36** *[real,parameter*=cos(36.0_cp**pi*/180.0_cp)*]*
- constants/**y10_norm** *[real]*

- constants/**cos72** *[real,parameter*=cos(72.0_cp**pi*/180.0_cp)*]*
- constants/**sin72** *[real,parameter*=sin(72.0_cp**pi*/180.0_cp)*]*
- constants/**c_dt_z10_ma** *[real]*
- constants/**two** *[real,parameter*=2.0_cp*]*
- constants/**half** *[real,parameter*=0.5_cp*]*
- constants/**vol_oc** *[real]*
  Volume of the outer core
- constants/**vol_ic** *[real]*
  Volume of the inner core
- constants/**y11_norm** *[real]*
- constants/**sq4pi** *[real,parameter*=sqrt(4.0_cp**pi*)*]*
- constants/**c_lorentz_ma** *[real]*
- constants/**third** *[real,parameter*=*one*/*three*]*
- constants/**codeversion** *[character,parameter*='5.3'*]*
- constants/**sin36** *[real,parameter*=sin(36.0_cp**pi*/180.0_cp)*]*
- constants/**mass** *[real]*
  Mass of the outer core

10.2.12 Grenoble.f90

Description

This module contains all variables for the case of an imposed IC dipole

Quick access

Variables *lgrenoble*, *bic*, *b0*, *db0*, *ddb0*

Routines *initialize_grenoble()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *truncation* (*n_r_maxmag()*): This module defines the grid points and the truncation

Variables

- grenoble/**b0** (:) *[real,allocatable/public]*
- grenoble/**ddb0** (:) *[real,allocatable/public]*
- grenoble/**db0** (:) *[real,allocatable/public]*
- grenoble/**lgrenoble** *[logical,public]*

- grenoble/**bic** *[real,public]*

Subroutines and functions

subroutine grenoble/**initialize_grenoble**()

Called from *magic*

10.3 MPI related modules

10.3.1 parallel.f90

Description

This module contains the blocking information

Quick access

Variables *chunksize*, *ierr*, *n_procs*, *nlmbs_per_rank*, *nr_on_last_rank*, *nr_per_rank*, *nthreads*, *rank_bn*, *rank_with_l1m0*

Routines *parallel()*

Needed modules

- omp_lib
- mpi

Variables

- parallel_mod/**nr_on_last_rank** *[integer]*
- parallel_mod/**rank_with_l1m0** *[integer]*
- parallel_mod/**nthreads** *[integer]*
- parallel_mod/**rank_bn** *[integer]*
- parallel_mod/**n_procs** *[integer]*
- parallel_mod/**nlmbs_per_rank** *[integer]*
- parallel_mod/**ierr** *[integer]*
- parallel_mod/**nr_per_rank** *[integer]*
- parallel_mod/**chunksize** *[integer]*

Subroutines and functions

subroutine parallel_mod/**parallel**()

Called from *magic*

10.3.2 radial_data.f90

Quick access

Variables *n_r_cmb*, *n_r_icb*, *nrstart*, *nrstartmag*, *nrstop*, *nrstopmag*

Routines *initialize_radial_data()*

Needed modules

- *parallel_mod* (*nr_on_last_rank()*, *nr_per_rank()*, *n_procs()*, rank()): This module contains the blocking information
- *logic* (*l_mag()*, *lverbose()*): Module containing the logicals that control the run
- *truncation* (*n_r_max()*): This module defines the grid points and the truncation

Variables

- radial_data/**n_r_cmb** *[integer,public]*
- radial_data/**n_r_icb** *[integer,public]*
- radial_data/**nrstopmag** *[integer,public]*
- radial_data/**nrstartmag** *[integer,public]*
- radial_data/**nrstop** *[integer,public]*
- radial_data/**nrstart** *[integer,public]*

Subroutines and functions

subroutine radial_data/**initialize_radial_data**()

Called from *magic*

10.3.3 LMLoop_data.f90

Quick access

Variables *llm*, *llmmag*, *lm_on_last_rank*, *lm_per_rank*, *ulm*, *ulmmag*

Routines *initialize_lmloop_data()*

Needed modules

- *parallel_mod* (*n_procs()*, *nlmbs_per_rank()*, rank()): This module contains the blocking information
- *blocking* (*sizelmb()*, *nlmbs()*, *lmstartb()*, *lmstopb()*): Module containing blocking information
- *logic* (*l_mag()*): Module containing the logicals that control the run

Variables

- lmloop_data/**llmmag** *[integer,public]*
- lmloop_data/**lm_per_rank** *[integer,public]*
- lmloop_data/**lm_on_last_rank** *[integer,public]*
- lmloop_data/**ulmmag** *[integer,public]*
- lmloop_data/**llm** *[integer,public]*
- lmloop_data/**ulm** *[integer,public]*

Subroutines and functions

subroutine lmloop_data/**initialize_lmloop_data**()

Called from *magic*

10.3.4 communications.f90

Quick access

Variables *get_global_sum*, *r_lm_gather_type*, *r_lm_gather_type_lm_end*, *final_wait_array*, *r_request*, *s_request*, *array_of_statuses*, *r_transfer_type*, *r_transfer_type_nr_end*, *s_transfer_type*, *s_transfer_type_nr_end*, *r_transfer_type_cont*, *r_transfer_type_nr_end_cont*, *s_transfer_type_cont*, *s_transfer_type_nr_end_cont*, *gt_cheb*, *gt_ic*, *gt_oc*, *lo2r_aj*, *lo2r_b*, *lo2r_p*, *lo2r_s*, *lo2r_w*, *lo2r_z*, *temp_gather_lo*, *temp_r2lo*

Routines *initialize_communications()*, *scatter_from_rank0_to_lo()*, *lm2lo_redist()*, *gather_from_lo_to_rank0()*, *lo2lm_redist()*, *r2lm_redist()*, *r2lo_redist()*, *destroy_gather_type()*, *destroy_lm2r_type()*, *lm2r_redist_wait()*, *lo2r_redist_wait()*, *lm2r_redist_start()*, *gather_all_from_lo_to_rank0()*, *lo2r_redist_start()*, *create_lm2r_type()*, *create_gather_type()*, *myallgather()*, *get_global_sum_cmplx_1d()*, *get_global_sum_cmplx_2d()*, *get_global_sum_real_2d()*

Needed modules

- *parallel_mod* (*nr_on_last_rank()*, *nr_per_rank()*, *n_procs()*, *ierr()*, rank()): This module contains the blocking information
- *lmloop_data* (*llm()*, *ulm()*)
- mpi
- *precision_mod*: This module controls the precision used in MagIC
- *logic* (*l_heat()*, *l_mag()*, *l_conv()*): Module containing the logicals that control the run
- *radial_data* (*nrstart()*, *nrstop()*)
- *blocking* (*lo_map()*, *lmstartb()*, *st_map()*, *lmstopb()*): Module containing blocking information
- *truncation* (*lm_max()*, *n_r_ic_max()*, *l_max()*, *minc()*, *n_r_max()*): This module defines the grid points and the truncation

Types

- **type** communications/**unknown_type**

  Type fields

  - **% count** *[integer]*
  - **% temp_rloc** (,,*) *[complex,pointer]*
  - **% r_request** (*) *[integer,allocatable]*
  - **% arr_rloc** (,,*) *[complex,pointer]*
  - **% final_wait_array** (*) *[integer,allocatable]*
  - **% s_request** (*) *[integer,allocatable]*

- **type** communications/**unknown_type**

  Type fields

  - **% dim2** *[integer]*
  - **% gather_mpi_type** (*) *[integer,allocatable]*

Variables

- communications/**s_transfer_type_cont** (:,:) *[integer,private/allocatable/save]*
- communications/**gt_ic** *[gather_type,public]*
- communications/**gt_oc** *[gather_type,public]*
- communications/**get_global_sum** *[public]*
- communications/**s_transfer_type** (:) *[integer,private/allocatable/save]*
- communications/**array_of_statuses** (:,:) *[integer,private/allocatable]*
- communications/**r_lm_gather_type** *[integer,private]*
- communications/**s_transfer_type_nr_end_cont** (:,:) *[integer,private/allocatable/save]*

- communications/**r_request** (:) *[integer,private/allocatable]*
- communications/**r_transfer_type** (:) *[integer,private/allocatable/save]*
- communications/**final_wait_array** (:) *[integer,private/allocatable]*
- communications/**s_request** (:) *[integer,private/allocatable]*
- communications/**r_transfer_type_cont** (:,:) *[integer,private/allocatable/save]*
- communications/**r_transfer_type_nr_end_cont** (:,:) *[integer,private/allocatable/save]*
- communications/**r_transfer_type_nr_end** (:) *[integer,private/allocatable/save]*
- communications/**lo2r_aj** *[lm2r_type,public]*
- communications/**s_transfer_type_nr_end** (:) *[integer,private/allocatable/save]*
- communications/**gt_cheb** *[gather_type,public]*
- communications/**r_lm_gather_type_lm_end** *[integer,private]*
- communications/**lo2r_z** *[lm2r_type,public]*
- communications/**lo2r_w** *[lm2r_type,public]*
- communications/**lo2r_p** *[lm2r_type,public]*
- communications/**lo2r_s** *[lm2r_type,public]*
- communications/**temp_r2lo** (:,:) *[complex,private/allocatable]*
- communications/**temp_gather_lo** (:) *[complex,private/allocatable]*
- communications/**lo2r_b** *[lm2r_type,public]*

Subroutines and functions

subroutine communications/**initialize_communications**()

Called from *magic*

Call to *create_gather_type()*, *create_lm2r_type()*

function communications/**get_global_sum_cmplx_2d**(*dwdt_local*)

Parameters **dwdt_local** (,) *[complex,in]*

Return **global_sum** *[real]*

function communications/**get_global_sum_real_2d**(*dwdt_local*)

Parameters **dwdt_local** (,) *[real,in]*

Return **global_sum** *[real]*

function communications/**get_global_sum_cmplx_1d**(*arr_local*)

Kahan summation algorithm

```
function KahanSum(input)
var sum = 0.0
var c = 0.0             //A running compensation for lost low-order bits.
for i = 1 to input.length do
   y = input[i] - c     //So far, so good: c is zero.
   t = sum + y          //Alas, sum is big, y small,
                        //so low-order digits of y are lost.
```

```
            c = (t - sum) - y   //(t - sum) recovers the high-order part of y;
                                //subtracting y recovers -(low part of y)
            sum = t             //Algebraically, c should always be zero.
                                //Beware eagerly optimising compilers!
            //Next time around, the lost low part will be added to y in a fresh attempt.
          return sum
```

Parameters **arr_local** (*) *[complex,in]*

Return **global_sum** *[real]*

subroutine communications/**gather_all_from_lo_to_rank0** (*self*, *arr_lo*, *arr_full*)

Parameters

- **self** *[gather_type]*
- **arr_lo** (*ulm*-(*llm*)+1,self%dim2) *[complex]*
- **arr_full** (*lm_max*,self%dim2) *[complex]*

Called from *fields_average()*, *output()*, *outpv()*, *get_dtblmfinish()*, *getegeos()*, *outto()*

subroutine communications/**create_gather_type** (*self*, *dim2*)

Define the datatypes for gather_all_from_lo_to_rank0 the sending array has dimension (llm:ulm,1:dim2) receiving array has dimension (1:lm_max,1:dim2)

Parameters

- **self** *[gather_type]*
- **dim2** *[integer]*

Called from *initialize_communications()*

subroutine communications/**destroy_gather_type** (*self*)

Parameters **self** *[gather_type]*

subroutine communications/**gather_from_lo_to_rank0** (*arr_lo*, *arr_full*)

Parameters

- **arr_lo** (*ulm*-(*llm*)+1) *[complex]*
- **arr_full** (*lm_max*) *[complex]*

Called from *fields_average()*, *storepotw()*

subroutine communications/**scatter_from_rank0_to_lo** (*arr_full*, *arr_lo*)

Parameters

- **arr_full** (*lm_max*) *[complex]*
- **arr_lo** (*ulm*-(*llm*)+1) *[complex]*

Called from *getstartfields()*

subroutine communications/**create_lm2r_type** (*self*[, *count*])

Parameters **self** *[lm2r_type]*

Options **count** *[integer,in,optional]*

Called from *initialize_communications()*

subroutine communications/**destroy_lm2r_type**(*self*)

Parameters **self** *[lm2r_type]*

subroutine communications/**lm2r_redist_start**(*self*, *arr_lmloc*, *arr_rloc*)

Parameters

- **self** *[lm2r_type]*
- **arr_lmloc** (*ulm*-(*llm*)+1,*n_r_max*,*) *[complex,in]*
- **arr_rloc** (*lm_max*,*nrstop*-(*nrstart*)+1,*) *[complex,out]*

Called from *lo2r_redist_start()*

subroutine communications/**lm2r_redist_wait**(*self*)

Parameters **self** *[lm2r_type]*

Called from *lo2r_redist_wait()*

subroutine communications/**lo2r_redist_start**(*self*, *arr_lo*, *arr_rloc*)

Parameters

- **self** *[lm2r_type]*
- **arr_lo** (*ulm*-(*llm*)+1,*n_r_max*,*) *[complex,in]*
- **arr_rloc** (*lm_max*,*nrstop*-(*nrstart*)+1,*) *[complex,out,target]*

Called from *lmloop()*, *getstartfields()*

Call to *lm2r_redist_start()*

subroutine communications/**lo2r_redist_wait**(*self*)

Parameters **self** *[lm2r_type]*

Called from *step_time()*

Call to *lm2r_redist_wait()*

subroutine communications/**r2lm_redist**(*arr_rloc*, *arr_lmloc*)

Parameters

- **arr_rloc** (*lm_max*,*nrstop*-(*nrstart*)+1) *[complex,in]*
- **arr_lmloc** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,out]*

Called from *r2lo_redist()*

subroutine communications/**r2lo_redist**(*arr_rloc*, *arr_lo*)

Parameters

- **arr_rloc** (*lm_max*,*nrstop*-(*nrstart*)+1) *[complex,in]*
- **arr_lo** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,out]*

Called from *step_time()*

Call to *r2lm_redist()*

subroutine communications/**lm2lo_redist**(*arr_lmloc*, *arr_lo*)

Parameters

- **arr_lmloc** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **arr_lo** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,out]*

subroutine `communications/`**`lo2lm_redist`**(*arr_lo*, *arr_lmloc*)

Parameters

- **arr_lo** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **arr_lmloc** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,out]*

subroutine `communications/`**`myallgather`**(*arr*, *dim1*, *dim2*)

Parameters

- **arr** (dim1,dim2) *[complex,inout]*
- **dim1** *[integer,in,]*
- **dim2** *[integer,in,]*

Use *parallel_mod*, *blocking*

Called from *dtvrms()*, *dtbrms()*

10.4 Code initialization

10.4.1 `Namelists.f90`

Description

Read and print the input namelist

Quick access

Routines *readnamelists()*, *defaultnamelists()*, *writenamelists()*

Needed modules

- *torsional_oscillations*: This module contains information for TO calculation and output
- *bext*: Module containing the external field parameters
- *movie_data* (*movie()*, *n_movies()*, *n_movies_max()*)
- *output_data*: This module contains the parameters for output control
- *logic*: Module containing the logicals that control the run
- *precision_mod*: This module controls the precision used in MagIC
- *init_fields*
- *parallel_mod*: This module contains the blocking information
- *num_param*: Module containing numerical and control parameters
- *physical_parameters*: Module containing the physical parameters

- *charmanip* (*capitalize()*, *length_to_blank()*): This module contains several useful routines to manipule character strings
- *truncation*: This module defines the grid points and the truncation
- *grenoble*: This module contains all variables for the case of an imposed IC dipole
- *blocking* (*cacheblock_size_in_b()*): Module containing blocking information
- *constants*: module containing constants and parameters used in the code.
- *radial_functions*: This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

Subroutines and functions

subroutine namelists/**readnamelists**()

Purpose of this subroutine is to read the input namelists. This program also determins logical parameters that are stored in logic.f90.

Called from *magic*

Call to *defaultnamelists()*, *length_to_blank()*, *capitalize()*, *initialize_truncation()*

subroutine namelists/**writenamelists**(*n_out*)

Purpose of this subroutine is to write the namelist to file unit n_out. This file has to be open before calling this routine.

Parameters **n_out** *[integer,in]*

Called from *magic*

Call to *length_to_blank()*

subroutine namelists/**defaultnamelists**()

Purpose of this subroutine is to set default parameters for the namelists.

Called from *readnamelists()*

10.4.2 startFiels.f90

Quick access

Routines *getstartfields()*

Needed modules

- *readcheckpoints* (*readstartfields()*): This module contains the functions that can help reading and mapping of the restart files
- *fieldslast*: This module contains time-derivaties array of the previous time-step They are needed in the time-stepping scheme.
- *useful* (*logwrite()*, *cc2real()*): library with several useful subroutines

- *communications* (*lo2r_z()*, *lo2r_w()*, *lo2r_p()*, *scatter_from_rank0_to_lo()*, *lo2r_s()*, *get_global_sum()*, *lo2r_aj()*, *lo2r_redist_start()*, *lo2r_b()*)
- *fields*: This module contains the potential fields and their radial derivatives
- mpi
- *logic* (*l_rot_ic()*, *l_z10mat()*, *l_lcr()*, *l_mag_kin()*, *l_mag()*, *l_heat()*, *l_srma()*, *l_mag_lf()*, *l_rot_ma()*, *l_sric()*, *l_cond_ic()*, *l_conv()*): Module containing the logicals that control the run
- *lmloop_data* (*llmmag()*, *lm_per_rank()*, *lm_on_last_rank()*, *ulm()*, *llm()*, *ulmmag()*)
- *radial_data* (*n_r_cmb()*, *n_r_icb()*)
- *precision_mod*: This module controls the precision used in MagIC
- *init_fields* (*s_cond()*, *initb()*, *init_b1()*, *l_start_file()*, *tops()*, *start_file()*, *initv()*, *inits()*, *init_s1()*)
- *radial_der_even* (*get_ddr_even()*)
- *parallel_mod* (*n_procs()*, *nlmbs_per_rank()*, rank()): This module contains the blocking information
- *num_param* (*alpha()*, *dtmax()*): Module containing numerical and control parameters
- *radial_der* (*get_dr()*, *get_ddr()*): Radial derivatives functions
- *physical_parameters* (*imps()*, *lffac()*, *kbotv()*, *n_r_lcr()*, *ktopv()*, *imagcon()*, *interior_model()*, *epss()*): Module containing the physical parameters
- *radial_functions* (*botcond()*, *drx()*, *ddrx()*, *chebt_oc()*, *chebt_ic_even()*, *or1()*, *topcond()*, *r()*, *dr_fac_ic()*, *chebt_ic()*, *dtemp0()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *grenoble* (*lgrenoble()*): This module contains all variables for the case of an imposed IC dipole
- *blocking* (*nlmbs()*, *lmstartb()*, *lo_map()*, *lmstopb()*): Module containing blocking information
- *constants* (*one()*, *osq4pi()*, *c_lorentz_ic()*, *two()*, *zero()*, *c_lorentz_ma()*): module containing constants and parameters used in the code.
- *truncation*: This module defines the grid points and the truncation

Variables

Subroutines and functions

subroutine start_fields/**getstartfields**(*time*, *dt*, *dtnew*, *n_time_step*)

Purpose of this subroutine is to initialize the fields and other auxiliary parameters.

Parameters

- **time** *[real,out]*
- **dt** *[real,out]*
- **dtnew** *[real,out]*
- **n_time_step** *[integer,out]*

Called from *magic*

Call to *s_cond()*, *readstartfields()*, *logwrite()*, *initb()*, *initv()*, *inits()*, *scatter_from_rank0_to_lo()*, *get_ddr()*, *get_ddr_even()*, *cc2real()*, *lo2r_redist_start()*

10.4.3 init_fields.f90

Quick access

Variables *init_b1*, *init_s1*, *init_s2*, *init_v1*, *nrotic*, *nrotma*, *n_start_file*, *inform*, *l_reset_t*, *l_start_file*, *n_s_bounds*, *start_file*, *amp_b1*, *amp_s1*, *amp_s2*, *amp_v1*, *bpeakbot*, *bpeaktop*, *omega_ic1*, *omega_ic2*, *omega_ma1*, *omega_ma2*, *omegaosz_ic1*, *omegaosz_ic2*, *omegaosz_ma1*, *omegaosz_ma2*, *scale_b*, *scale_s*, *scale_v*, *tomega_ic1*, *tomega_ic2*, *tomega_ma1*, *tomega_ma2*, *tshift_ic1*, *tshift_ic2*, *tshift_ma1*, *tshift_ma2*, *tipdipole*, *bots*, *tops*, *s_top*, *s_bot*

Routines *initialize_init_fields()*, *initb()*, *j_cond()*, *inits()*, *s_cond()*, *initv()*

Needed modules

- *horizontal_data* (*hdif_b()*, *d_lp1()*, *dlh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *legendre_grid_to_spec* (*legtf1()*)
- *cosine_transform_odd*
- *useful* (*random()*): library with several useful subroutines
- *algebra* (*sgesl()*, *sgefa()*, *cgesl()*)
- *fft*: This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5
- *lmloop_data* (*ulmmag()*, *llmmag()*, *llm()*, *ulm()*)
- *radial_data* (*n_r_cmb()*, *n_r_icb()*)
- *precision_mod*: This module controls the precision used in MagIC
- *logic* (*l_rot_ic()*, *l_anelastic_liquid()*, *l_cond_ic()*, *l_srma()*, *l_rot_ma()*, *l_sric()*): Module containing the logicals that control the run
- *physical_parameters* (*imps()*, *n_imps()*, *kbots()*, *peaks()*, *sigma_ratio()*, *thetas()*, *widths()*, *radratio()*, *opr()*, *epsc()*, *phis()*, *imagcon()*, *ktops()*, *o_sr()*, *opm()*, *n_imps_max()*): Module containing the physical parameters
- *truncation*: This module defines the grid points and the truncation
- *matrices* (*jmat()*, *s0mat()*, *jpivot()*, *s0pivot()*): This module contains matricies for internal time step
- *blocking* (*st_map()*, *nthetabs()*, *nfs()*, *lmp2lmps()*, *sizethetab()*): Module containing blocking information
- *constants* (*osq4pi()*, *third()*, *y10_norm()*, *c_z10_omega_ma()*, *two()*, *one()*, *four()*, *zero()*, *c_z10_omega_ic()*, *three()*, *half()*, *pi()*): module containing constants and parameters used in the code.

- *radial_functions* (*jvarcon()*, *or2()*, *or3()*, *or1()*, *cheb()*, *otemp1()*, *orho1()*, *d2cheb_ic()*, *dlkappa()*, *chebt_oc()*, *cheb_norm_ic()*, *dllambda()*, *dcheb_ic()*, *cheb_norm()*, *cheb_ic()*, *beta()*, *dcheb()*, *chebt_ic()*, *dtemp0()*, *d2cheb()*, *kappa()*, *r_cmb()*, *epscprof()*, *r_ic()*, *r()*, *r_icb()*, *lambda()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

- init_fields/**tshift_ma1** *[real,public]*
- init_fields/**tshift_ma2** *[real,public]*
- init_fields/**amp_b1** *[real,public]*
- init_fields/**s_bot** (80) *[real,public]*
  input variables for tops,bots
- init_fields/**init_b1** *[integer,public]*
- init_fields/**omegaosz_ic2** *[real,public]*
- init_fields/**n_s_bounds** *[integer,parameter*=20/public*]*
- init_fields/**omegaosz_ic1** *[real,public]*
- init_fields/**n_start_file** *[integer,public]*
  I/O unit of start_file
- init_fields/**tomega_ma1** *[real,public]*
- init_fields/**tomega_ma2** *[real,public]*
- init_fields/**omega_ic2** *[real,public]*
- init_fields/**scale_b** *[real,public]*
- init_fields/**l_start_file** *[logical,public]*
  taking fields from startfile ?
- init_fields/**omega_ic1** *[real,public]*
- init_fields/**start_file** *[character,public]*
  name of start_file
- init_fields/**scale_s** *[real,public]*
- init_fields/**init_v1** *[integer,public]*
- init_fields/**scale_v** *[real,public]*
- init_fields/**amp_v1** *[real,public]*
- init_fields/**init_s1** *[integer,public]*
- init_fields/**tipdipole** *[real,public]*
  adding to symetric field
- init_fields/**tomega_ic2** *[real,public]*
- init_fields/**tomega_ic1** *[real,public]*
- init_fields/**bpeaktop** *[real,public]*
- init_fields/**tops** (:,:) *[complex,allocatable/public]*
- init_fields/**nrotma** *[integer,public]*

- init_fields/**omega_ma1** *[real,public]*
- init_fields/**omega_ma2** *[real,public]*
- init_fields/**init_s2** *[integer,public]*
- init_fields/**tshift_ic2** *[real,public]*
- init_fields/**tshift_ic1** *[real,public]*
- init_fields/**s_top** (80) *[real,public]*
- init_fields/**inform** *[integer,public]*
  format of start_file
- init_fields/**bpeakbot** *[real,public]*
- init_fields/**amp_s2** *[real,public]*
- init_fields/**omegaosz_ma1** *[real,public]*
- init_fields/**omegaosz_ma2** *[real,public]*
- init_fields/**amp_s1** *[real,public]*
- init_fields/**l_reset_t** *[logical,public]*
  reset time from startfile ?
- init_fields/**bots** (:,:) *[complex,allocatable/public]*
- init_fields/**nrotic** *[integer,public]*

Subroutines and functions

subroutine init_fields/**initialize_init_fields**()

Called from *magic*

subroutine init_fields/**initv**(*w*, *z*, *omega_ic*, *omega_ma*, *lmstart*, *lmstop*)

Purpose of this subroutine is to initialize the velocity field So far it is only rudimentary and will be expanded later. Because s is needed for dwdt init_s has to be called before.

Parameters

- **w** (*lm_max*,*n_r_max*) *[complex,inout]*
- **z** (*lm_max*,*n_r_max*) *[complex,inout]*
- **omega_ic** *[real,out]*
- **omega_ma** *[real,out]*
- **lmstart** *[integer,in]*
- **lmstop** *[integer,in]*

Called from *getstartfields()*

Call to *fft_thetab()*, *legtf1()*, *random()*

subroutine init_fields/**inits**(*s*, *lmstart*, *lmstop*)

Purpose of this subroutine is to initialize the entropy field according to the input control parameters.

Input	value
init_s1 < 100:	random noise initialized the noise spectrum decays as l ^ (init_s1-1) with peak amplitude amp_s1 for l=1
init_s1 >=100:	a specific harmonic mode initialized with amplitude amp_s1. init_s1 is interpreted as number llmm where ll: harmonic degree, mm: harmonic order.
init_s2 >100 :	a second harmonic mode initialized with amplitude amp_s2. init_s2 is again interpreted as number llmm where ll: harmonic degree, mm: harmonic order.

Parameters

- **s** (*lm_max*,*n_r_max*) *[complex,inout]*
- **lmstart** *[integer,in]*
- **lmstop** *[integer,in]*

Called from *getstartfields()*

Call to *s_cond()*, *random()*, *fft_thetab()*, *legtf1()*, *sgefa()*, *sgesl()*

subroutine init_fields/**initb**(*b*, *aj*, *b_ic*, *aj_ic*, *lorentz_torque_ic*, *lorentz_torque_ma*, *lmstart*, *lmstop*)

Parameters

- **b** (*lm_maxmag*,*n_r_maxmag*) *[complex,inout]*
- **aj** (*lm_maxmag*,*n_r_maxmag*) *[complex,inout]*
- **b_ic** (*lm_maxmag*,*n_r_ic_max*) *[complex,inout]*
- **aj_ic** (*lm_maxmag*,*n_r_ic_max*) *[complex,inout]*
- **lorentz_torque_ic** *[real,out]*
- **lorentz_torque_ma** *[real,out]*
- **lmstart** *[integer,in]*
- **lmstop** *[integer,in]*

Called from *getstartfields()*

Call to *j_cond()*, *random()*

subroutine init_fields/**j_cond**(*lm0*, *aj0*, *aj0_ic*)

Purpose of this subroutine is to solve the diffusion equation for an initial toroidal magnetic field.

Parameters

- **lm0** *[integer,in]*
- **aj0** (*) *[complex,out]*
- **aj0_ic** (*) *[complex,out]*

Called from *initb()*

Call to *sgefa()*, *cgesl()*

subroutine init_fields/**s_cond**(*s0*)

Purpose of this subroutine is to solve the entropy equation for an the conductive (l=0,m=0)-mode. Output is the radial dependence of the solution in s0.

Parameters **s0** (*) *[real,out]*

Called from *inits()*, *getstartfields()*

Call to *sgefa()*, *sgesl()*

10.5 Pre-calculations

10.5.1 preCalc.f90

Quick access

Routines *precalc()*, *writeinfo()*, *get_hit_times()*, *precalctimes()*

Needed modules

- *horizontal_data* (*horizontal()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *parallel_mod* (rank()): This module contains the blocking information
- *bext* (*fac_loop()*, *l_curr()*): Module containing the external field parameters
- *useful* (*logwrite()*): library with several useful subroutines
- *output_data*: This module contains the parameters for output control
- *logic* (*l_time_hits()*, *l_non_rot()*, *l_anelastic_liquid()*, *l_tomovie()*, *l_cmb_field()*, *l_dt_cmb_field()*, *l_newmap()*, *l_storetpot()*, *l_storepot()*, *l_lcr()*, *l_cond_ic()*, *l_mag_lf()*, *l_storevpot()*, *l_anel()*, *l_heat()*, *l_to()*, *l_storebpot()*, *l_save_out()*, *l_mag()*, *l_movie()*, *l_r_field()*): Module containing the logicals that control the run
- *integration* (*rint_r()*): Radial integration functions
- *precision_mod*: This module controls the precision used in MagIC
- *init_fields* (*s_top()*, *tops()*, *s_bot()*, *l_reset_t()*, *n_s_bounds()*, *bots()*)
- *num_param*: Module containing numerical and control parameters
- *physical_parameters* (*polind()*, *lffac()*, *ek()*, *epsc0()*, *tmagcon()*, *ktops()*, *rho_ratio_ic()*, *pr()*, *sigma_ratio()*, *epsc()*, *ra()*, *n_r_lcr()*, *o_sr()*, *rho_ratio_ma()*, *nvardiff()*, *buofac()*, *kbots()*, *opr()*, *radratio()*, *nvareps()*, *opm()*, *nvarvisc()*, *prmag()*, *r_lcr()*, *corfac()*, *mode()*, *interior_model()*, *nvarcond()*, *ekscaled()*, *rascaled()*): Module containing the physical parameters
- *radial_functions* (*ddrx()*, *rho0()*, *dlvisc()*, *temp0()*, *dlkappa()*, *transportproperties()*, *chebt_oc()*, *dddrx()*, *dllambda()*, *rgrav()*, *r_surface()*, *visc()*, *beta()*, *divktemp0()*, *drx()*, *kappa()*, *r_cmb()*, *dbeta()*, *r()*, *radial()*, *r_icb()*, *dentropy0()*, *sigma()*, *lambda()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *constants*: module containing constants and parameters used in the code.

- *truncation*(*n_r_ic_max()*, *n_cheb_ic_max()*, *minc()*, *n_phi_max()*, *l_max()*, *nalias()*, *n_r_max()*, *m_max()*, *lm_max()*, *n_cheb_max()*, *n_theta_max()*): This module defines the grid points and the truncation

Variables

Subroutines and functions

subroutine precalculations/**precalc**()

Purpose of this subroutine is to initialize the calc values, arrays, constants that are used all over the code. The stuff is stored in the common blocks. MPI: This is called by every processors.

Called from *magic*

Call to *radial()*, *transportproperties()*, *horizontal()*, *rint_r()*, *logwrite()*

subroutine precalculations/**precalctimes**(*time*, *n_time_step*)

Precalc. after time, time and dthas been read from startfile.

Parameters

- **time** *[real,out]*
- **n_time_step** *[integer,out]*

Called from *magic*

Call to *get_hit_times()*

subroutine precalculations/**get_hit_times**(*t*, *n_t_max*, *n_t*, *l_t*, *t_start*, *t_stop*, *dt*, *n_tot*, *n_step*, *string_bn*, *time*, *tscale*)

Parameters

- **t** (n_t_max) *[real,inout]* :: Times for output
- **n_t_max** *[integer,in,]* :: Dimension of t(*)
- **n_t** *[integer,out]* :: No. of output times
- **l_t** *[logical,out]* :: =.true. if output times are defined
- **t_start** *[real,inout]* :: Starting time for output
- **t_stop** *[real,inout]* :: Stop time for output
- **dt** *[real,inout]* :: Time step for output
- **n_tot** *[integer,inout]* :: No. of output (times) if no times defined
- **n_step** *[integer,inout]* :: Ouput step in no. of time steps
- **string_bn** *[character,in]*
- **time** *[real,in]* :: Time of start file
- **tscale** *[real,in]*

Called from *precalctimes()*

subroutine precalculations/**writeinfo**(*n_out*)

Purpose of this subroutine is to write the namelist to file unit n_out. This file has to be open before calling this routine.

Parameters **n_out** *[integer,in]*

Called from *magic*

10.5.2 radial.f90

Description

This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Quick access

Variables *ndd_costf1*, *ndd_costf1_ic*, *ndd_costf2_ic*, *ndi_costf1*, *ndi_costf1_ic*, *ndi_costf2_ic*, *chebt_ic*, *chebt_ic_even*, *chebt_oc*, *dr_fac*, *cheb_norm*, *cheb_norm_ic*, *dr_fac_ic*, *botcond*, *topcond*, *r_icb*, *alph2*, *alpha2*, *alph1*, *alpha1*, *r_cmb*, *r_surface*, *agrav*, *beta*, *cheb_int*, *cheb_int_ic*, *d2temp0*, *dbeta*, *dddrx*, *ddrx*, *dentropy0*, *divktemp0*, *dlkappa*, *dllambda*, *dlvisc*, *drx*, *dtemp0*, *epscprof*, *jvarcon*, *kappa*, *lambda*, *o_r_ic*, *o_r_ic2*, *or1*, *or2*, *or3*, *or4*, *orho1*, *orho2*, *otemp1*, *r*, *r_ic*, *rgrav*, *rho0*, *sigma*, *temp0*, *visc*, *cheb*, *cheb_ic*, *d2cheb*, *d2cheb_ic*, *d3cheb*, *dcheb*, *dcheb_ic*

Routines *initialize_radial_functions()*, *transportproperties()*, *radial()*, *getbackground()*

Needed modules

- *cosine_transform_odd*
- *algebra* (*sgesl()*, *sgefa()*)
- *logic* (*l_anelastic_liquid()*, *l_newmap()*, *l_mag()*, *l_heat()*, *l_anel()*, *l_cond_ic()*, *l_isothermal()*): Module containing the logicals that control the run
- *cosine_transform_even*
- *num_param* (*alpha()*): Module containing numerical and control parameters
- *chebyshev_polynoms_mod*
- *physical_parameters*: Module containing the physical parameters
- *matrices* (*s0mat()*, *s0pivot()*): This module contains matricies for internal time step
- *radial_der* (*get_dr()*): Radial derivatives functions
- *constants* (*four()*, *three()*, *two()*, *sq4pi()*, *half()*, *one()*): module containing constants and parameters used in the code.
- *truncation* (*n_r_ic_max()*, *n_cheb_max()*, *n_r_max()*): This module defines the grid points and the truncation

Variables

- radial_functions/**ddrx** (:) *[real,allocatable/public]*
- radial_functions/**jvarcon** (:) *[real,allocatable/public]*
- radial_functions/**dlvisc** (:) *[real,allocatable/public]*
- radial_functions/**dentropy0** (:) *[real,allocatable/public]*
- radial_functions/**otemp1** (:) *[real,allocatable/public]*
- radial_functions/**dlkappa** (:) *[real,allocatable/public]*
- radial_functions/**chebt_oc** *[costf_odd_t,public]*
- radial_functions/**dr_fac** *[real,public]*
  $2/d$, where $d = r_o - r_i$
- radial_functions/**ndi_costf2_ic** *[integer,public]*
  Radii for transform
- radial_functions/**dddrx** (:) *[real,allocatable/public]*
- radial_functions/**dllambda** (:) *[real,allocatable/public]*
- radial_functions/**dcheb_ic** (:,:) *[real,allocatable/public]*
- radial_functions/**alph2** *[real,public]*
  Input parameter for non-linear map to define central point of different spacing (-1.0:1.0)
- radial_functions/**cheb_int_ic** (:) *[real,allocatable/public]*
- radial_functions/**orho1** (:) *[real,allocatable/public]*
- radial_functions/**orho2** (:) *[real,allocatable/public]*
- radial_functions/**dr_fac_ic** *[real,public]*
  For IC: $2/(2r_i)$
- radial_functions/**o_r_ic2** (:) *[real,allocatable/public]*
- radial_functions/**dcheb** (:,:) *[real,allocatable/public]*
- radial_functions/**dtemp0** (:) *[real,allocatable/public]*
- radial_functions/**d2cheb** (:,:) *[real,allocatable/public]*
- radial_functions/**agrav** (:) *[real,allocatable/public]*
- radial_functions/**kappa** (:) *[real,allocatable/public]*
- radial_functions/**r_cmb** *[real,public]*
  OC radius
- radial_functions/**epscprof** (:) *[real,allocatable/public]*
- radial_functions/**dbeta** (:) *[real,allocatable/public]*
- radial_functions/**ndd_costf2_ic** *[integer,public]*
  Radii for transform
- radial_functions/**topcond** *[real,public]*
  Heat flux at OC boundary
- radial_functions/**r** (:) *[real,allocatable/public]*

- radial_functions/**r_icb** *[real,public]*
  IC radius
- radial_functions/**alph1** *[real,public]*
  Input parameter for non-linear map to define degree of spacing (0.0:2.0)
- radial_functions/**or4** (:) *[real,allocatable/public]*
- radial_functions/**rho0** (:) *[real,allocatable/public]*
- radial_functions/**or2** (:) *[real,allocatable/public]*
- radial_functions/**or3** (:) *[real,allocatable/public]*
- radial_functions/**or1** (:) *[real,allocatable/public]*
- radial_functions/**temp0** (:) *[real,allocatable/public]*
- radial_functions/**cheb** (:,:) *[real,allocatable/public]*
- radial_functions/**o_r_ic** (:) *[real,allocatable/public]*
- radial_functions/**ndi_costf1** *[integer,public]*
  Radii for transform
- radial_functions/**alpha2** *[real,public]*
  Input parameter for non-linear map to define central point of different spacing (-1.0:1.0)
- radial_functions/**alpha1** *[real,public]*
  Input parameter for non-linear map to define degree of spacing (0.0:2.0)
- radial_functions/**ndd_costf1** *[integer,public]*
  Radii for transform
- radial_functions/**cheb_norm_ic** *[real,public]*
  Chebyshev normalisation for IC
- radial_functions/**rgrav** (:) *[real,allocatable/public]*
- radial_functions/**cheb_int** (:) *[real,allocatable/public]*
- radial_functions/**r_surface** *[real,public]*
  Surface radius for extrapolation
- radial_functions/**d2cheb_ic** (:,:) *[real,allocatable/public]*
- radial_functions/**cheb_norm** *[real,public]*
  Chebyshev normalisation
- radial_functions/**cheb_ic** (:,:) *[real,allocatable/public]*
- radial_functions/**ndi_costf1_ic** *[integer,public]*
  Radii for transform
- radial_functions/**visc** (:) *[real,allocatable/public]*
- radial_functions/**beta** (:) *[real,allocatable/public]*
- radial_functions/**chebt_ic_even** *[costf_even_t,public]*
- radial_functions/**chebt_ic** *[costf_odd_t,public]*
- radial_functions/**d3cheb** (:,:) *[real,allocatable/public]*
- radial_functions/**divktemp0** (:) *[real,allocatable/public]*
- radial_functions/**botcond** *[real,public]*
  Heat flux at IC boundary

- radial_functions/**drx** (:) *[real,allocatable/public]*
- radial_functions/**d2temp0** (:) *[real,allocatable/public]*
- radial_functions/**r_ic** (:) *[real,allocatable/public]*
- radial_functions/**ndd_costf1_ic** *[integer,public]*
  Radii for transform
- radial_functions/**sigma** (:) *[real,allocatable/public]*
- radial_functions/**lambda** (:) *[real,allocatable/public]*

Subroutines and functions

subroutine radial_functions/**initialize_radial_functions**()
Initial memory allocation

Called from *magic*

subroutine radial_functions/**radial**()
Calculates everything needed for radial functions, transforms etc.

Called from *precalc()*

Call to *cheb_grid()*, *getbackground()*, *get_chebs_even()*

subroutine radial_functions/**transportproperties**()
Calculates the transport properties: electrical conductivity, kinematic viscosity and thermal conductivity.

Called from *precalc()*

subroutine radial_functions/**getbackground**(*input*, *boundaryval*, *output*)

Linear solver of the form: df/dx = input with f(1)=boundaryVal

Parameters

- **input** (*n_r_max*) *[real,in]*
- **boundaryval** *[real,in]*
- **output** (*n_r_max*) *[real,out]*

Called from *radial()*

Call to *sgefa()*, *sgesl()*

10.5.3 horizontal.f90

Description

Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

Quick access

Variables *lmodd*, *lmoddp*, *lstart*, *lstartp*, *lstop*, *lstopp*, *n_theta_cal2ord*, *cosn2*, *costheta*, *d_l*, *d_lp1*, *d_m*, *d_mc2m*, *dlh*, *dphi*, *dphi0*, *dphi02*, *dpl0eq*, *dtheta1a*, *dtheta1s*, *dtheta2a*, *dtheta2s*, *dtheta3a*, *dtheta3s*, *dtheta4a*, *dtheta4s*,

gauss, *hdif_b*, *hdif_s*, *hdif_v*, *o_sin_theta*, *o_sin_theta_e2*, *osn1*, *osn2*, *phi*, *sintheta*, *sn2*, *theta*, *theta_ord*, *dplm*, *plm*, *wplm*

Routines *initialize_horizontal_data()*, *horizontal()*, *gauleg()*

Needed modules

- *fft*: This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5
- *num_param* (*ldif()*, *ldifexp()*, *difeta()*, *difkap()*, *difnu()*): Module containing numerical and control parameters
- *precision_mod*: This module controls the precision used in MagIC
- *plms_theta* (*plm_theta()*, *plm_thetaas()*)
- *logic* (*l_non_rot()*): Module containing the logicals that control the run
- *physical_parameters* (*ek()*): Module containing the physical parameters
- *truncation* (*l_max()*, *minc()*, *n_phi_max()*, *lmp_max()*, *n_m_max()*, *m_max()*, *lm_max()*, *n_theta_max()*): This module defines the grid points and the truncation
- *blocking* (*lmp2l()*, *lm2m()*, *lm2l()*, *lmp2lm()*): Module containing blocking information
- *constants* (*zero()*, *pi()*, *two()*, *half()*, *one()*): module containing constants and parameters used in the code.
- *radial_functions* (*r_cmb()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

- horizontal_data/**dplm** (:,:) *[real,allocatable/public]*
- horizontal_data/**dphi02** (:) *[complex,allocatable/public]*
- horizontal_data/**osn1** (:) *[real,allocatable/public]*
- horizontal_data/**osn2** (:) *[real,allocatable/public]*
- horizontal_data/**sn2** (:) *[real,allocatable/public]*
- horizontal_data/**o_sin_theta** (:) *[real,allocatable/public]*
- horizontal_data/**dphi** (:) *[complex,allocatable/public]*
- horizontal_data/**lstopp** (:) *[integer,allocatable/public]*
- horizontal_data/**plm** (:,:) *[real,allocatable/public]*
- horizontal_data/**costheta** (:) *[real,allocatable/public]*
- horizontal_data/**hdif_b** (:) *[real,allocatable/public]*
- horizontal_data/**lstop** (:) *[integer,allocatable/public]*
- horizontal_data/**dphi0** (:) *[complex,allocatable/public]*
- horizontal_data/**dtheta4a** (:) *[real,allocatable/public]*
- horizontal_data/**lmodd** (:) *[logical,allocatable/public]*
- horizontal_data/**dlh** (:) *[real,allocatable/public]*

- horizontal_data/**dtheta3s** (:) *[real,allocatable/public]*
- horizontal_data/**dtheta2a** (:) *[real,allocatable/public]*
- horizontal_data/**hdif_s** (:) *[real,allocatable/public]*
- horizontal_data/**hdif_v** (:) *[real,allocatable/public]*
- horizontal_data/**dtheta4s** (:) *[real,allocatable/public]*
- horizontal_data/**theta** (:) *[real,allocatable/public]*
- horizontal_data/**dtheta2s** (:) *[real,allocatable/public]*
- horizontal_data/**lstart** (:) *[integer,allocatable/public]*
- horizontal_data/**phi** (:) *[real,allocatable/public]*
- horizontal_data/**theta_ord** (:) *[real,allocatable/public]*
- horizontal_data/**d_mc2m** (:) *[real,allocatable/public]*
- horizontal_data/**o_sin_theta_e2** (:) *[real,allocatable/public]*
- horizontal_data/**gauss** (:) *[real,allocatable/public]*
- horizontal_data/**wplm** (:,:) *[real,allocatable/public]*
- horizontal_data/**d_lp1** (:) *[real,allocatable/public]*
- horizontal_data/**lstartp** (:) *[integer,allocatable/public]*
- horizontal_data/**dtheta3a** (:) *[real,allocatable/public]*
- horizontal_data/**cosn2** (:) *[real,allocatable/public]*
- horizontal_data/**dtheta1a** (:) *[real,allocatable/public]*
- horizontal_data/**dpl0eq** (:) *[real,allocatable/public]*
- horizontal_data/**lmoddp** (:) *[logical,allocatable/public]*
- horizontal_data/**d_l** (:) *[real,allocatable/public]*
- horizontal_data/**d_m** (:) *[real,allocatable/public]*
- horizontal_data/**dtheta1s** (:) *[real,allocatable/public]*
- horizontal_data/**n_theta_cal2ord** (:) *[integer,allocatable/public]*
- horizontal_data/**sintheta** (:) *[real,allocatable/public]*

Subroutines and functions

subroutine horizontal_data/**initialize_horizontal_data**()

Called from *magic*

subroutine horizontal_data/**horizontal**()

Calculates functions of theta and phi, for exmample the Legendre functions, and functions of degree l and order m of the legendres.

Called from *precalc()*

Call to *gauleg()*, *plm_theta()*, *plm_thetaas()*, *init_fft()*

subroutine horizontal_data/**gauleg**(*sinthmin*, *sinthmax*, *theta_ord*, *gauss*, *n_th_max*)

Subroutine is based on a NR code. Calculates N zeros of legendre polynomial P(l=N) in the interval [sinThMin,sinThMax]. Zeros are returned in radiants theta_ord(i) The respective weights for Gauss-integration are given in gauss(i).

Parameters

- **sinthmin** *[real,in]* :: lower bound in radiants
- **sinthmax** *[real,in]* :: upper bound in radiants
- **theta_ord** (n_th_max) *[real,out]* :: zeros cos(theta)
- **gauss** (n_th_max) *[real,out]* :: associated Gauss-Legendre weights
- **n_th_max** *[integer,in]* :: desired maximum degree

Called from *horizontal()*

10.6 Time stepping

10.6.1 step_time.f90

Quick access

Variables *dbdt_lmloc*, *djdt_lmloc*, *djdt_rloc*, *dpdt_lmloc*, *dpdt_rloc*, *dsdt_lmloc*, *dsdt_rloc*, *dvsrlm_lmloc*, *dvsrlm_rloc*, *dvxbhlm_lmloc*, *dvxbhlm_rloc*, *dwdt_lmloc*, *dwdt_rloc*, *dzdt_lmloc*, *dzdt_rloc*, *dbdt_rloc*

Routines *initialize_step_time()*, *check_time_hits()*, *step_time()*

Needed modules

- *movie_data* (*t_movies()*)
- *output_mod* (*output()*)
- *lmloop_mod* (*lmloop()*)
- *lmloop_data* (*llmmag()*, *lm_per_rank()*, *lm_on_last_rank()*, *ulm()*, *llm()*, *ulmmag()*)
- *courant_mod* (*dt_courant()*)
- *useful* (*safeclose()*, *l_correct_step()*, *safeopen()*, *logwrite()*): library with several useful subroutines
- *radialloop* (*radialloopg()*)
- *parallel_mod*: This module contains the blocking information
- *constants* (*zero()*, *half()*, *one()*): module containing constants and parameters used in the code.
- *num_param* (*tscale()*, *n_time_steps()*, *tend()*, *runtimelimit()*, *alpha()*, *dtmax()*, *dtmin()*, *runtime()*): Module containing numerical and control parameters
- *logic* (*l_time_hits()*, *l_hel()*, *l_to()*, *l_dtrmagspec()*, *l_storetpot()*, *lverbose()*, *l_heat()*, *l_mag_lf()*, *l_perppar()*, *l_b_nl_icb()*, *l_tomovie()*, *l_mag()*, *l_dtbmovie()*, *l_htmovie()*, *l_storevpot()*, *l_runtimelimit()*, *l_fluxprofs()*, *l_storebpot()*, *l_viscbccalc()*, *l_conv()*, *l_movie()*, *l_cmb_field()*,

l_true_time(), *l_ht()*, *l_dtb()*, *l_save_out()*, *l_b_nl_cmb()*, *l_rms()*, *l_r_field()*): Module containing the logicals that control the run

- *precision_mod*: This module controls the precision used in MagIC
- *timing*: Useful functions for time-stepping
- *charmanip* (*dble2str()*, *capitalize()*): This module contains several useful routines to manipule character strings
- *blocking* (*nlmbs()*, *lmstartb()*, *lmstopb()*): Module containing blocking information
- *nonlinear_bcs* (*get_b_nl_bcs()*)
- *truncation* (*l_max()*, *lm_maxmag()*, *lmp_max()*, *l_maxmag()*, *n_r_max()*, *lm_max()*, *n_r_maxmag()*): This module defines the grid points and the truncation
- *fieldslast*: This module contains time-derivaties array of the previous time-step They are needed in the time-stepping scheme.
- *output_data* (*n_cmb_step()*, *n_t_to()*, *n_tozs()*, *n_graphs()*, *n_tomovie_step()*, *n_r_field_step()*, *n_vpots()*, *tag()*, *n_tpots()*, *n_cmbs()*, *n_t_cmb()*, *nlf()*, *t_to()*, *n_logs()*, *t_cmb()*, *n_tomovie_frames()*, *n_bpot_step()*, *t_log()*, *n_tos()*, *n_tpot_step()*, *n_t_bpot()*, *n_rsts()*, *t_graph()*, *n_t_graph()*, *n_t_rst()*, *n_t_toz()*, *n_toz_step()*, *t_toz()*, *n_rst_step()*, *n_spec_step()*, *graph_mpi_fh()*, *n_stores()*, *n_t_vpot()*, *t_bpot()*, *n_t_spec()*, *n_log_file()*, *n_time_hits()*, *n_movie_frames()*, *graph_file()*, *n_specs()*, *t_tomovie()*, *n_graph_step()*, *t_spec()*, *n_bpots()*, *t_tpot()*, *t_movie()*, *n_t_tpot()*, *n_t_tomovie()*, *n_to_step()*, *n_t_log()*, *t_vpot()*, *l_graph_time()*, *n_log_step()*, *n_r_fields()*, *n_movie_step()*, *t_r_field()*, *n_vpot_step()*, *n_t_r_field()*, *t_rst()*, *log_file()*, *n_t_movie()*): This module contains the parameters for output control
- *radial_data* (*n_r_cmb()*, *n_r_icb()*, *nrstopmag()*, *nrstartmag()*, *nrstop()*, *nrstart()*)
- *communications* (*lo2r_redist_wait()*, lm2r_type(), *lo2r_z()*, *lo2r_w()*, *lo2r_p()*, *lo2r_s()*, *r2lo_redist()*, *get_global_sum()*, *lo2r_aj()*, *lo2r_redist_start()*, *lo2r_b()*)
- *fields*: This module contains the potential fields and their radial derivatives

Variables

- step_time_mod/**dbdt_lmloc** (:,:) *[complex,private/allocatable]*
- step_time_mod/**dzdt_lmloc** (:,:) *[complex,private/allocatable]*
- step_time_mod/**dvsrlm_lmloc** (:,:) *[complex,private/allocatable]*
- step_time_mod/**dbdt_rloc** (:,:) *[complex,private/target/allocatable]*
- step_time_mod/**dpdt_rloc** (:,:) *[complex,private/allocatable]*
- step_time_mod/**dwdt_rloc** (:,:) *[complex,private/allocatable]*
- step_time_mod/**dsdt_rloc** (:,:) *[complex,private/allocatable]*
- step_time_mod/**djdt_rloc** (:,:) *[complex,private/allocatable]*
- step_time_mod/**dsdt_lmloc** (:,:) *[complex,private/allocatable]*
- step_time_mod/**dvxbhlm_rloc** (:,:) *[complex,private/allocatable]*

- step_time_mod/**dzdt_rloc** (:,:) *[complex,private/allocatable]*
- step_time_mod/**dpdt_lmloc** (:,:) *[complex,private/allocatable]*
- step_time_mod/**dvxbhlm_lmloc** (:,:) *[complex,private/allocatable]*
- step_time_mod/**dwdt_lmloc** (:,:) *[complex,private/allocatable]*
- step_time_mod/**dvsrlm_rloc** (:,:) *[complex,private/allocatable]*
- step_time_mod/**djdt_lmloc** (:,:) *[complex,private/allocatable]*

Subroutines and functions

subroutine step_time_mod/**initialize_step_time**()

Called from *magic*

subroutine step_time_mod/**step_time**(*time*, *dt*, *dtnew*, *n_time_step*)

This subroutine performs the actual time-stepping.

Parameters

- **time** *[real,inout]*
- **dt** *[real,inout]*
- **dtnew** *[real,inout]*
- **n_time_step** *[integer,inout]*

Called from *magic*

Call to *walltime()*, *lo2r_redist_wait()*, *capitalize()*, *time2ms()*, *ms2time()*, *ltimelimit()*, *logwrite()*, *l_correct_step()*, *dble2str()*, *safeopen()*, *safeclose()*, *radialloopg()*, *lnegtime()*, *subtime()*, *addtime()*, *r2lo_redist()*, *output()*, *get_b_nl_bcs()*, *dt_courant()*, *check_time_hits()*, *lmloop()*, *meantime()*, *writetime()*

subroutine step_time_mod/**check_time_hits**(*l_new_dt*, *time*, *dt*, *dt_new*)

Checks whether a certain dt is required to hit a specific output-time.

Parameters

- **l_new_dt** *[logical,out]* :: signfies change of dt !
- **time** *[real,inout]*
- **dt** *[real,inout]*
- **dt_new** *[real,inout]*

Called from *step_time()*

10.6.2 courant.f90

Quick access

Routines *dt_courant()*, *courant()*

Needed modules

- *horizontal_data* (*osn2()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *precision_mod*: This module controls the precision used in MagIC
- *useful* (*logwrite()*): library with several useful subroutines
- *logic* (*l_mag()*, *l_mag_kin()*, *l_mag_lf()*): Module containing the logicals that control the run
- *radial_data* (*nrstart()*, *nrstop()*)
- *parallel_mod*: This module contains the blocking information
- *num_param* (*delxh2()*, *delxr2()*, *courfac()*, *alffac()*): Module containing numerical and control parameters
- *physical_parameters* (*opm()*, *lffac()*): Module containing the physical parameters
- *truncation* (*nrp()*, *n_phi_max()*): This module defines the grid points and the truncation
- *blocking* (*nfs()*): Module containing blocking information
- *constants* (*one()*, *two()*, *half()*): module containing constants and parameters used in the code.
- *radial_functions* (*or2()*, *or4()*, *orho1()*, *orho2()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

Subroutines and functions

subroutine courant_mod/**courant** (*n_r*, *dtrkc*, *dthkc*, *vr*, *vt*, *vp*, *br*, *bt*, *bp*, *n_theta_min*, *n_theta_block*)

Parameters

- **n_r** *[integer,in]* :: radial level
- **dtrkc** *[real,inout]* :: Courant step (based on radial advection)
- **dthkc** *[real,inout]* :: Courant step based on horizontal advection
- **vr** (*nrp*,*nfs*) *[real,in]* :: radial velocity
- **vt** (*nrp*,*nfs*) *[real,in]* :: longitudinal velocity
- **vp** (*nrp*,*nfs*) *[real,in]* :: azimuthal velocity
- **br** (*nrp*,*nfs*) *[real,in]* :: radial magnetic field
- **bt** (*nrp*,*nfs*) *[real,in]* :: longitudinal magnetic field
- **bp** (*nrp*,*nfs*) *[real,in]* :: azimuthal magnetic field
- **n_theta_min** *[integer,in]* :: first theta in block stored in fields
- **n_theta_block** *[integer,in]* :: size of theta block

Called from *do_iteration_thetablocking_openmp()*, do_iteration_thetablocking_seq(), do_iteration_thetablocking_shtns()

subroutine courant_mod/**dt_courant** (*dt_r*, *dt_h*, *l_new_dt*, *dt*, *dt_new*, *dtmax*, *dtrkc*, *dthkc*)

Check if Courant criterion based on combined fluid and Alfven velocity is satisfied Returns new value of time step dtnew

dtr,dth: (output) radial/horizontal Courant time step n_time_step: (input) time step number l_new_dt: (output) flag indicating that time step is changed (=1) or not (=0) dt: (input) old time step dtnew: (output) new time step dtMin: (input) lower limit for time step (termination if dtnew < dtMin) dtMax: (input) upper limit for time step dtrkc: (input) radial Courant time step as function of radial level dthkc: (input) horizontal Courant time step as function of radial level

Parameters

- **dt_r** *[real,out]*
- **dt_h** *[real,out]*
- **l_new_dt** *[logical,out]*
- **dt** *[real,in]*
- **dt_new** *[real,out]*
- **dtmax** *[real,in]*
- **dtrkc** (*nrstop*-(*nrstart*)+1) *[real,in]*
- **dthkc** (*nrstop*-(*nrstart*)+1) *[real,in]*

Called from *step_time()*

Call to *logwrite()*

10.6.3 timing.f90

Description

Useful functions for time-stepping

Quick access

Variables *msecsecond*, *msechour*, *msecminute*

Routines *ms2time()*, *writetime()*, *walltime()*, *meantime()*, *addtime()*, *subtime()*, *time2ms()*, *ltimelimit()*, *lnegtime()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod* (rank()): This module contains the blocking information
- mpi

Variables

- timing/**msechour** *[integer,private/parameter=3600000]*
- timing/**msecsecond** *[integer,private/parameter=1000]*

- timing/**msecminute** *[integer,private/parameter=60000]*

Subroutines and functions

subroutine timing/**walltime**(*time*)

This routine returns the wallclock time in four integer arguments.

Parameters **time** (4) *[integer,out]*

Called from *step_time()*, *lmloop()*, *magic*

Call to *ms2time()*

subroutine timing/**ms2time**(*ms*, *time*)

Transforms accumulated milliseconds ms into an four-element integer arrays time(4) containing the time in hours=time(1), minutes=time(2), seconds=time(3), and milliseconds=time(4).

Parameters

- **ms** *[integer,in]*
- **time** (4) *[integer,out]*

Called from *step_time()*, *meantime()*, *addtime()*, *subtime()*, *walltime()*

function timing/**time2ms**(*time*)

Transforms a four-element integer arrays time(4) containing the time in hours=time(1), minutes=time(2), seconds=time(3), and milliseconds=time(4) into accumulated milliseconds.

Parameters **time** (4) *[integer,in]*

Return **time2ms** *[integer]*

Called from *step_time()*, *meantime()*, *lnegtime()*, *addtime()*, *subtime()*

subroutine timing/**subtime**(*timestart*, *timestop*, *timed*)

Returns time passed between timeStop and timeStart. Note timeStop has to be younger than timeStart, otherwise 24 hours are added. This is necessary on systems like the IBM where the time counter as reset every day at midnight.

Parameters

- **timestart** (4) *[integer,in]*
- **timestop** (4) *[integer,in]*
- **timed** (4) *[integer,out]*

Called from *step_time()*, *lmloop()*

Call to *time2ms()*, *ms2time()*

function timing/**ltimelimit**(*time*, *timemax*)

True when time exeeds timeMax

Parameters

- **time** (4) *[integer,in]*
- **timemax** (4) *[integer,in]*

Return **ltimelimit** *[logical]*

Called from *step_time()*

subroutine timing/**addtime**(*time1*, *time2*)

Parameters

- **time1** (4) *[integer]*
- **time2** (4) *[integer]*

Called from *step_time()*

Call to *time2ms()*, *ms2time()*

subroutine timing/**meantime**(*time*, *n*)

Parameters

- **time** (4) *[integer]*
- **n** *[integer,in]*

Called from *step_time()*

Call to *time2ms()*, *ms2time()*

function timing/**lnegtime**(*time1*, *time2*)

Negative passed time? Means we have passed midnight. The wallclock time is reset to zero on some computers at midnight.

Parameters

- **time1** (4) *[integer,in]*
- **time2** (4) *[integer,in]*

Return **lnegtime** *[logical]*

Called from *step_time()*

Call to *time2ms()*

subroutine timing/**writetime**(*nout*, *text*, *time*)

Returns time passed between timeStop and timeStart. Note timeStop has to be younger than timeStart, otherwise 24 hours are added. This is necessary on systems like the IBM where the time counter are reset every day at midnight.

Parameters

- **nout** *[integer,in]*
- **text** *[character,in]*
- **time** (4) *[integer,in]*

Called from *step_time()*, *lmloop()*, *magic*

10.7 Linear calculation part of the time stepping (LMLoop)

10.7.1 LMLoop.f90

Quick access

Routines *initialize_lmloop()*, *lmloop()*

Needed modules

- debugging (debug_write())
- *parallel_mod* (rank()): This module contains the blocking information
- *updatewp_mod* (*updatewp()*, *initialize_updatewp()*)
- *output_data* (*log_file()*, *nlf()*): This module contains the parameters for output control
- *useful* (*safeclose()*, *safeopen()*): library with several useful subroutines
- *fieldslast*: This module contains time-derivaties array of the previous time-step They are needed in the time-stepping scheme.
- *updates_mod* (*initialize_updates()*, *updates_ala()*, *updates()*)
- *fields*: This module contains the potential fields and their radial derivatives
- *updatez_mod* (*initialize_updatez()*, *updatez()*)
- *lmloop_data* (*ulmmag()*, *llmmag()*, *llm()*, *ulm()*)
- *radial_data* (*n_r_cmb()*, *n_r_icb()*)
- *precision_mod*: This module controls the precision used in MagIC
- *communications* (*lo2r_z()*, *lo2r_w()*, *lo2r_p()*, *lo2r_s()*, *get_global_sum()*, *lo2r_aj()*, *lo2r_redist_start()*, *lo2r_b()*)
- *updateb_mod* (*initialize_updateb()*, *updateb()*)
- *logic* (*l_heat()*, *l_anelastic_liquid()*, *l_mag()*, *lverbose()*, *l_conv()*): Module containing the logicals that control the run
- *timing* (*writetime()*, *subtime()*, *walltime()*): Useful functions for time-stepping
- omp_lib
- *matrices* (*lbmat()*, *lz10mat()*, *lwpmat()*, *lsmat()*, *lzmat()*): This module contains matricies for internal time step
- *blocking* (*lmstartb()*, *lmstopb()*): Module containing blocking information
- *truncation* (*lm_max()*, *l_max()*, *n_r_max()*, *n_r_maxmag()*): This module defines the grid points and the truncation

Variables

Subroutines and functions

subroutine lmloop_mod/**initialize_lmloop**()

Called from *magic*

Call to *initialize_updates()*, *initialize_updatez()*, *initialize_updatewp()*, *initialize_updateb()*

subroutine lmloop_mod/**lmloop**(*w1*, *coex*, *time*, *dt*, *lmat*, *lrmsnext*, *dvxbhlm*, *dvsrlm*, *dsdt*, *dwdt*, *dzdt*, *dpdt*, *dbdt*, *djdt*, *lorentz_torque_ma*, *lorentz_torque_ic*, *b_nl_cmb*, *aj_nl_cmb*, *aj_nl_icb*)

This subroutine performs the actual time-stepping.

Parameters

- **w1** *[real,in]*
- **coex** *[real,in]*
- **time** *[real,in]*
- **dt** *[real,in]*
- **lmat** *[logical,in]*
- **lrmsnext** *[logical,in]*
- **dvxbhlm** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,inout]*
- **dvsrlm** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,inout]*
- **dsdt** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,inout]*
- **dwdt** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **dzdt** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **dpdt** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **dbdt** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,in]*
- **djdt** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,inout]*
- **lorentz_torque_ma** *[real,in]*
- **lorentz_torque_ic** *[real,in]*
- **b_nl_cmb** (*lm_max*) *[complex,in]* :: nonlinear bc for b at CMB
- **aj_nl_cmb** (*lm_max*) *[complex,in]* :: nonlinear bc for aj at CMB
- **aj_nl_icb** (*lm_max*) *[complex,in]* :: nonlinear bc for dr aj at ICB

Called from *step_time()*

Call to *safeopen()*, *walltime()*, *updates_ala()*, *updates()*, *lo2r_redist_start()*, *updatewp()*, *updateb()*, *subtime()*, *writetime()*, *safeclose()*

10.7.2 updateWP.f90

Quick access

Variables *buo*, *pre*

Routines *finalize_updatewp()*, *initialize_updatewp()*, *get_wpmat()*, *get_p0mat()*, *updatewp()*

Needed modules

- *horizontal_data* (*dlh()*, *hdif_v()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *rms* (*difpol2hint()*, *dtvpollmr()*, *dtvpol2hint()*, *difpollmr()*): This module contains the calculation of thr RMS force balance and induction terms.
- *truncation* (*lm_max()*, *n_cheb_max()*, *n_r_max()*): This module defines the grid points and the truncation
- *algebra* (*sgesl()*, *sgefa()*, *cgeslml()*)
- *cosine_transform_odd*
- *rms_helpers* (*hint2pol()*): This module contains several useful subroutines required to compute RMS diagnostics
- *logic* (*l_update_v()*): Module containing the logicals that control the run
- *lmloop_data* (*llm()*, *ulm()*)
- *radial_data* (*n_r_cmb()*, *n_r_icb()*)
- *precision_mod*: This module controls the precision used in MagIC
- *communications* (*get_global_sum()*)
- *parallel_mod* (*chunksize()*): This module contains the blocking information
- *num_param* (*alpha()*): Module containing numerical and control parameters
- *radial_der* (*get_dr()*, *get_dddr()*): Radial derivatives functions
- *physical_parameters* (*kbotv()*, *ktopv()*, *ra()*): Module containing the physical parameters
- omp_lib
- *matrices* (*p0mat()*, *wppivot()*, *wpmat()*, *p0pivot()*, *lwpmat()*, *wpmat_fac()*): This module contains matricies for internal time step
- *blocking* (*lo_sub_map()*, *lmstartb()*, *nlmbs()*, *lmstopb()*, *lo_map()*, *st_map()*, *st_sub_map()*): Module containing blocking information
- *constants* (*third()*, *three()*, *two()*, *four()*, *zero()*, *half()*, *one()*): module containing constants and parameters used in the code.
- *radial_functions* (*d2cheb()*, *drx()*, *ddrx()*, *cheb_norm()*, *chebt_oc()*, *rho0()*, *or2()*, *agrav()*, *or1()*, *visc()*, *beta()*, *dlvisc()*, *cheb()*, *rgrav()*, *dcheb()*, *dddrx()*, *d3cheb()*, *dbeta()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

- updatewp_mod/**pre** (:) *[complex,private/allocatable]*
- updatewp_mod/**maxthreads** *[integer,private]*
- updatewp_mod/**worka** (:,:) *[complex,private/allocatable]*
- updatewp_mod/**workb** (:,:) *[complex,private/allocatable]*
- updatewp_mod/**rhs1** (:,:,:) *[complex,private/allocatable]*
- updatewp_mod/**buo** (:) *[complex,private/allocatable]*

- updatewp_mod/**dif** (:) *[complex,private/allocatable]*
- updatewp_mod/**dtv** (:) *[complex,private/allocatable]*

Subroutines and functions

subroutine updatewp_mod/**initialize_updatewp**()

Called from *initialize_lmloop()*

subroutine updatewp_mod/**finalize_updatewp**()

subroutine updatewp_mod/**updatewp**(*w*, *dw*, *ddw*, *dwdt*, *dwdtlast*, *p*, *dp*, *dpdt*, *dpdtlast*, *s*, *w1*, *coex*, *dt*, *nlmb*, *lrmsnext*)

updates the poloidal velocity potential w, the pressure p, and their derivatives adds explicit part to time derivatives of w and p

Parameters

- **w** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,inout]*
- **dw** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,inout]*
- **ddw** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,out]*
- **dwdt** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **dwdtlast** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,inout]*
- **p** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,inout]*
- **dp** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,out]*
- **dpdt** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **dpdtlast** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,inout]*
- **s** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **w1** *[real,in]* :: weight for time step !
- **coex** *[real,in]* :: factor depending on alpha
- **dt** *[real,in]* :: time step
- **nlmb** *[integer,in]* :: block number
- **lrmsnext** *[logical,in]*

Called from *lmloop()*

Call to *get_p0mat()*, *get_wpmat()*, *sgesl()*, *cgeslml()*, *get_dddr()*, *hint2pol()*

subroutine updatewp_mod/**get_wpmat**(*dt*, *l*, *hdif*, *wpmat*, *wppivot*, *wpmat_fac*)

Purpose of this subroutine is to contruct the time step matrix wpmat for the NS equation.

Parameters

- **dt** *[real,in]*
- **l** *[integer,in]*
- **hdif** *[real,in]*

- **wpmat** (2*n_r_max,2*n_r_max) *[real,out]*
- **wppivot** (2*n_r_max) *[integer,out]*
- **wpmat_fac** (2*n_r_max,2) *[real,out]*

Called from updatewp()

Call to sgefa()

subroutine updatewp_mod/**get_p0mat** (*pmat*, *ppivot*)

Parameters

- **pmat** (n_r_max,n_r_max) *[real,out]*
- **ppivot** (n_r_max) *[integer,out]*

Called from updatewp()

Call to sgefa()

10.7.3 updateZ.f90

Quick access

Variables maxthreads, dif, dtv, worka, workb, workc, rhs1

Routines initialize_updatez(), get_z10mat(), get_zmat(), updatez()

Needed modules

- horizontal_data (dlh(), hdif_v()): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- rms (dtvtor2hint(), diftor2hint()): This module contains the calculation of thr RMS force balance and induction terms.
- lmloop_data (llm(), ulm())
- init_fields
- radial_der (get_ddr()): Radial derivatives functions
- cosine_transform_odd
- parallel_mod: This module contains the blocking information
- num_param (alpha(), amstart()): Module containing numerical and control parameters
- physical_parameters (kbotv(), ktopv(), lffac()): Module containing the physical parameters
- omp_lib
- radial_functions (d2cheb(), drx(), ddrx(), cheb_norm(), r_cmb(), chebt_oc(), rho0(), or2(), or1(), visc(), beta(), dlvisc(), cheb(), r_icb(), dcheb(), r(), dbeta()): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- algebra (cgesl(), sgefa(), cgeslml())
- torsional_oscillations (ddzasl()): This module contains information for TO calculation and output

- *rms_helpers* (*hint2tor()*): This module contains several useful subroutines required to compute RMS diagnostics
- *matrices* (*lz10mat()*, *zmat_fac()*, *zmat()*, *z10mat()*, *z10mat_fac()*, *z10pivot()*, *zpivot()*, *lzmat()*): This module contains matricies for internal time step
- *blocking* (*lo_sub_map()*, *lmstartb()*, *nlmbs()*, *lmstopb()*, *lo_map()*, *st_map()*, *st_sub_map()*): Module containing blocking information
- *constants* (*c_moi_oc()*, *y11_norm()*, *y10_norm()*, *c_z10_omega_ma()*, *c_lorentz_ic()*, *c_dt_z10_ic()*, *one()*, *four()*, *c_moi_ma()*, *zero()*, *c_z10_omega_ic()*, *two()*, *c_moi_ic()*, *half()*, *c_dt_z10_ma()*, *c_lorentz_ma()*): module containing constants and parameters used in the code.
- *truncation* (*lm_max()*, *l_max()*, *n_cheb_max()*, *n_r_max()*): This module defines the grid points and the truncation
- *precision_mod*: This module controls the precision used in MagIC
- *outrot* (*get_angular_moment()*)
- *radial_data* (*n_r_cmb()*, *n_r_icb()*)
- *communications* (*get_global_sum()*)
- *logic* (*l_rot_ic()*, *l_to()*, *l_z10mat()*, *l_correct_amz()*, *l_correct_ame()*, *l_update_v()*, *l_srma()*, *l_rot_ma()*, *l_sric()*): Module containing the logicals that control the run

Variables

- updatez_mod/**maxthreads** *[integer,private]*
- updatez_mod/**worka** (:,:) *[complex,private/allocatable]*
- updatez_mod/**workb** (:,:) *[complex,private/allocatable]*
- updatez_mod/**workc** (:,:) *[complex,private/allocatable]*
- updatez_mod/**rhs1** (:,:,:) *[complex,private/allocatable]*
- updatez_mod/**dtv** (:) *[complex,private/allocatable]*
- updatez_mod/**dif** (:) *[complex,private/allocatable]*

Subroutines and functions

subroutine updatez_mod/**initialize_updatez**()

Called from *initialize_lmloop()*

subroutine updatez_mod/**updatez**(*z*, *dz*, *dzdt*, *dzdtlast*, *time*, *omega_ma*, *d_omega_ma_dtlast*, *omega_ic*, *d_omega_ic_dtlast*, *lorentz_torque_ma*, *lorentz_torque_malast*, *lorentz_torque_ic*, *lorentz_torque_iclast*, *w1*, *coex*, *dt*, *lrmsnext*)

updates the toroidal potential z and its radial derivatives adds explicit part to time derivatives of z

Parameters

- **z** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,inout]*
- **dz** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,out]*

- **dzdt** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **dzdtlast** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,inout]*
- **time** *[real,in]* :: Current time
- **omega_ma** *[real,out]* :: Calculated OC rotation
- **d_omega_ma_dtlast** *[real,inout]* :: Time derivative of OC rotation of previous step
- **omega_ic** *[real,out]* :: Calculated IC rotation
- **d_omega_ic_dtlast** *[real,inout]* :: Time derivative of IC rotation of previous step
- **lorentz_torque_ma** *[real,in]* :: Lorentz torque (for OC rotation)
- **lorentz_torque_malast** *[real,in]* :: Lorentz torque (for OC rotation) of previous step
- **lorentz_torque_ic** *[real,in]* :: Lorentz torque (for IC rotation)
- **lorentz_torque_iclast** *[real,in]* :: Lorentz torque (for IC rotation) of previous step
- **w1** *[real,in]* :: Weight for time step
- **coex** *[real,in]* :: Factor depending on alpha
- **dt** *[real,in]* :: Time step interval
- **lrmsnext** *[logical,in]* :: Logical for storing update if (l_RMS.and.l_logNext)

Call to *get_zmat()*, *get_z10mat()*, *cgesl()*, *cgesl1()*, *get_ddr()*, *get_angular_moment()*, *hint2tor()*

subroutine updatez_mod/**get_z10mat** (*dt*, *l*, *hdif*, *zmat*, *zpivot*, *zmat_fac*)

Purpose of this subroutine is to construct and LU-decompose the inversion matrix z10mat for the implicit time step of the toroidal velocity potential z of degree l=1 and order m=0. This differs from the the normal zmat only if either the ICB or CMB have no-slip boundary condition and inner core or mantle are chosen to rotate freely (either kbotv=1 and/or ktopv=1).

Parameters

- **dt** *[real,in]* :: Time step internal
- **l** *[integer,in]* :: Variable to loop over l's
- **hdif** *[real,in]* :: Value of hyperdiffusivity in zMat terms
- **zmat** (*n_r_max*,*n_r_max*) *[real,out]* :: LHS matrix to calculate z
- **zpivot** (*n_r_max*) *[integer,out]* :: Pivot to invert zMat
- **zmat_fac** (*n_r_max*) *[real,out]* :: Inverse of max(zMat) for inversion

Called from *updatez()*

Call to *sgefa()*

subroutine updatez_mod/**get_zmat** (*dt*, *l*, *hdif*, *zmat*, *zpivot*, *zmat_fac*)

Purpose of this subroutine is to contruct the time step matricies zmat(i,j) for the NS equation.

Parameters

- **dt** *[real,in]* :: Time interval
- **l** *[integer,in]* :: Variable to loop over degrees

- **hdif** *[real,in]* :: Hyperdiffusivity
- **zmat** (*n_r_max*,*n_r_max*) *[real,out]* :: Matrix with LHS of z equation
- **zpivot** (*n_r_max*) *[integer,out]* :: Pivot for zMat inversion
- **zmat_fac** (*n_r_max*) *[real,out]* :: Inverse of max(zMat) for the inversion

Called from *updatez()*

Call to *sgefa()*

10.7.4 updateS.f90

Quick access

Routines *initialize_updates()*, *get_smat()*, *get_s0mat()*, *updates()*, *updates_ala()*

Needed modules

- *horizontal_data* (*hdif_s()*, *dlh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *parallel_mod* (*chunksize()*, rank()): This module contains the blocking information
- *truncation* (*lm_max()*, *n_cheb_max()*, *n_r_max()*): This module defines the grid points and the truncation
- *algebra* (*sgesl()*, *sgefa()*, *cgeslml()*)
- *cosine_transform_odd*
- *logic* (*l_anelastic_liquid()*, *l_update_s()*): Module containing the logicals that control the run
- *lmloop_data* (*llm()*, *ulm()*)
- *radial_data* (*n_r_cmb()*, *n_r_icb()*)
- *precision_mod*: This module controls the precision used in MagIC
- *init_fields* (*bots()*, *tops()*)
- *num_param* (*alpha()*): Module containing numerical and control parameters
- *radial_der* (*get_drns()*, *get_ddr()*): Radial derivatives functions
- *physical_parameters* (*kbots()*, *ktops()*, *opr()*): Module containing the physical parameters
- omp_lib
- *matrices* (*smat()*, *smat_fac()*, *s0mat()*, *lsmat()*, *s0mat_fac()*, *s0pivot()*, *spivot()*): This module contains matricies for internal time step
- *blocking* (*lo_sub_map()*, *lmstartb()*, *nlmbs()*, *lmstopb()*, *lo_map()*, *st_map()*): Module containing blocking information
- *constants* (*zero()*, *half()*, *two()*, *one()*): module containing constants and parameters used in the code.

- *radial_functions* (*d2cheb()*, *drx()*, *ddrx()*, *orho1()*, *dlkappa()*, *chebt_oc()*, *or2()*, *or1()*, *beta()*, *temp0()*, *dentropy0()*, *kappa()*, *dcheb()*, *otemp1()*, *cheb_norm()*, *dtemp0()*, *cheb()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

- updates_mod/**maxthreads** *[integer,private]*
- updates_mod/**worka** (:,:) *[complex,private/allocatable]*
- updates_mod/**workb** (:,:) *[complex,private/allocatable]*
- updates_mod/**rhs1** (:,:,:) *[complex,private/allocatable]*

Subroutines and functions

subroutine updates_mod/**initialize_updates**()

Called from *initialize_lmloop()*

subroutine updates_mod/**updates**(*s*, *ds*, *dvsrlm*, *dsdt*, *dsdtlast*, *w1*, *coex*, *dt*, *nlmb*)

updates the entropy field s and its radial derivatives adds explicit part to time derivatives of s

Parameters

- **s** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,inout]*
- **ds** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,out]*
- **dvsrlm** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,inout]*
- **dsdt** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,inout]*
- **dsdtlast** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,inout]*
- **w1** *[real,in]* :: weight for time step !
- **coex** *[real,in]* :: factor depending on alpha
- **dt** *[real,in]* :: time step
- **nlmb** *[integer,in]*

Called from *lmloop()*

Call to *get_drns()*, *get_s0mat()*, *get_smat()*, *sgesl()*, *cgeslml()*, *get_ddr()*

subroutine updates_mod/**updates_ala**(*s*, *ds*, *w*, *dvsrlm*, *dsdt*, *dsdtlast*, *w1*, *coex*, *dt*, *nlmb*)

updates the entropy field s and its radial derivatives adds explicit part to time derivatives of s

Parameters

- **s** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,inout]*
- **ds** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,out]*
- **w** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **dvsrlm** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,inout]*
- **dsdt** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,inout]*

- **dsdtlast** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,inout]*
- **w1** *[real,in]* :: weight for time step !
- **coex** *[real,in]* :: factor depending on alpha
- **dt** *[real,in]* :: time step
- **nlmb** *[integer,in]*

Called from `lmloop()`

Call to `get_drns()`, `get_s0mat()`, `get_smat()`, `sgesl()`, `cgeslml()`, `get_ddr()`

subroutine updates_mod/**get_s0mat**(*dt*, *smat*, *spivot*, *smat_fac*)

Purpose of this subroutine is to contruct the time step matrix sMat0

Parameters

- **dt** *[real,in]*
- **smat** (*n_r_max*,*n_r_max*) *[real,out]*
- **spivot** (*n_r_max*) *[integer,out]*
- **smat_fac** (*n_r_max*) *[real,out]*

Called from `updates()`, `updates_ala()`

Call to `sgefa()`

subroutine updates_mod/**get_smat**(*dt*, *l*, *hdif*, *smat*, *spivot*, *smat_fac*)

Purpose of this subroutine is to contruct the time step matricies sMat(i,j) and s0mat for the entropy equation.

Parameters

- **dt** *[real,in]*
- **l** *[integer,in]*
- **hdif** *[real,in]*
- **smat** (*n_r_max*,*n_r_max*) *[real,out]*
- **spivot** (*n_r_max*) *[integer,out]*
- **smat_fac** (*n_r_max*) *[real,out]*

Called from `updates()`, `updates_ala()`

Call to `sgefa()`

10.7.5 updateB.f90

Quick access

Variables `dtp`, `dtt`, `rhs2`

Routines `initialize_updateb()`, `updateb()`, `get_bmat()`

Needed modules

- *horizontal_data* (*hdif_b()*, *dphi()*, *dlh()*, *d_l()*, *d_lp1()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *rms* (*dtbtor2hint()*, *dtbpollmr()*, *dtbpol2hint()*): This module contains the calculation of thr RMS force balance and induction terms.
- *lmloop_data* (*llmmag()*, *ulmmag()*)
- *init_fields* (*bpeakbot()*, *bpeaktop()*)
- *radial_der_even* (*get_ddr_even()*)
- *radial_der* (*get_drns()*, *get_ddr()*): Radial derivatives functions
- *cosine_transform_odd*
- *parallel_mod* (*chunksize()*, rank()): This module contains the blocking information
- *num_param* (*alpha()*): Module containing numerical and control parameters
- *physical_parameters* (*sigma_ratio()*, *ktopb()*, *tmagcon()*, *imagcon()*, *o_sr()*, *n_r_lcr()*, *opm()*, *conductance_ma()*, *kbotb()*): Module containing the physical parameters
- omp_lib
- *radial_functions* (*d2cheb_ic()*, *drx()*, *ddrx()*, *cheb_norm()*, *dcheb_ic()*, *r_cmb()*, *chebt_oc()*, *cheb_norm_ic()*, *or2()*, *dr_fac_ic()*, *or1()*, *dllambda()*, *r()*, *cheb()*, *o_r_ic()*, *chebt_ic_even()*, *dcheb()*, *chebt_ic()*, *cheb_ic()*, *d2cheb()*, *lambda()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *algebra* (*sgefa()*, *cgeslml()*)
- *precision_mod*: This module controls the precision used in MagIC
- *rms_helpers* (*hint2torlm()*, *hint2pollm()*): This module contains several useful subroutines required to compute RMS diagnostics
- *matrices* (*jpivot()*, *lbmat()*, *bpivot()*, *bmat_fac()*, *bmat()*, *jmat_fac()*, *jmat()*): This module contains matricies for internal time step
- *blocking* (*lo_sub_map()*, *lmstartb()*, *nlmbs()*, *lmstopb()*, *lo_map()*, *st_map()*, *st_sub_map()*): Module containing blocking information
- *constants* (*one()*, *three()*, *two()*, *zero()*, *half()*, *pi()*): module containing constants and parameters used in the code.
- *truncation* (*n_r_ic_max()*, *n_cheb_ic_max()*, *n_r_ic_maxmag()*, *n_r_tot()*, *lm_max()*, *n_r_max()*, *n_r_totmag()*, *n_cheb_max()*, *n_r_maxmag()*): This module defines the grid points and the truncation
- *bext*: Module containing the external field parameters
- *radial_data* (*n_r_cmb()*, *n_r_icb()*)
- *logic* (*l_rot_ic()*, *l_mag_nl()*, *l_lcr()*, *l_cond_ic()*, *l_b_nl_cmb()*, *l_b_nl_icb()*, *l_update_b()*): Module containing the logicals that control the run

Variables

- updateb_mod/**maxthreads** *[integer,private]*
- updateb_mod/**worka** (:,:) *[complex,private/allocatable]*

- updateb_mod/**workb** (:,:) *[complex,private/allocatable]*
- updateb_mod/**rhs2** (:,:,:) *[complex,private/allocatable]*
- updateb_mod/**rhs1** (:,:,:) *[complex,private/allocatable]*
- updateb_mod/**dtp** (:) *[complex,private/allocatable]*
- updateb_mod/**dtt** (:) *[complex,private/allocatable]*

Subroutines and functions

subroutine updateb_mod/**initialize_updateb**()

Called from *initialize_lmloop()*

subroutine updateb_mod/**updateb**(*b*, *db*, *ddb*, *aj*, *dj*, *ddj*, *dvxbhlm*, *dbdt*, *dbdtlast*, *djdt*, *djdtlast*, *b_ic*, *db_ic*, *ddb_ic*, *aj_ic*, *dj_ic*, *ddj_ic*, *dbdt_iclast*, *djdt_iclast*, *b_nl_cmb*, *aj_nl_cmb*, *aj_nl_icb*, *omega_ic*, *w1*, *coex*, *dt*, *time*, *nlmb*, *lrmsnext*)

Calculated update of magnetic field potential and the time stepping arrays dbdtLast, ...

updates the magnetic field potentials b, aj and their derivatives, adds explicit part to time derivatives of b and j

Parameters

- **b** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,inout]*
- **db** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,out]*
- **ddb** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,out]*
- **aj** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,inout]*
- **dj** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,out]*
- **ddj** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,out]*
- **dvxbhlm** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,inout]*
- **dbdt** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,in]*
- **dbdtlast** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,inout]*
- **djdt** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,inout]*
- **djdtlast** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,inout]*
- **b_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,inout]*
- **db_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,out]*
- **ddb_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,out]*
- **aj_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,inout]*
- **dj_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,out]*
- **ddj_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,out]*
- **dbdt_iclast** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,inout]*
- **djdt_iclast** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,inout]*
- **b_nl_cmb** (*) *[complex,in]*
- **aj_nl_cmb** (*) *[complex,in]*

- **aj_nl_icb** (*) *[complex,in]*
- **omega_ic** *[real,in]*
- **w1** *[real,in]* :: weight for time step !
- **coex** *[real,in]* :: factor depending on alpha
- **dt** *[real,in]*
- **time** *[real,in]*
- **nlmb** *[integer,in]*
- **lrmsnext** *[logical,in]*

Called from *lmloop()*

Call to *get_drns()*, *get_bmat()*, *cgeslml()*, *get_ddr()*, *get_ddr_even()*, *hint2pollm()*, *hint2torlm()*

subroutine updateb_mod/**get_bmat**(*dt*, *l*, *hdif*, *bmat*, *bpivot*, *bmat_fac*, *jmat*, *jpivot*, *jmat_fac*)

Purpose of this subroutine is to contruct the time step matrices bmat(i,j) and ajmat for the dynamo equations.

Parameters

- **dt** *[real,in]*
- **l** *[integer,in]*
- **hdif** *[real,in]*
- **bmat** (*n_r_totmag*,*n_r_totmag*) *[real,out]*
- **bpivot** (*n_r_totmag*) *[integer,out]*
- **bmat_fac** (*n_r_totmag*) *[real,out]*
- **jmat** (*n_r_totmag*,*n_r_totmag*) *[real,out]*
- **jpivot** (*n_r_totmag*) *[integer,out]*
- **jmat_fac** (*n_r_totmag*) *[real,out]*

Called from *updateb()*

Call to *sgefa()*

10.8 Non-linear part of the time stepping (radial loop)

10.8.1 radialLoop.f90

Quick access

Routines *finalize_radialloop()*, *initialize_radialloop()*, *radialloopg()*

Needed modules

- *parallel_mod* (*n_procs()*, rank()): This module contains the blocking information
- *graphout_mod* (*graphout_mpi_header()*)
- *riterthetablocking_mod* (riterthetablocking_t())
- *fft*: This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5
- *radial_data* (*n_r_cmb()*, *n_r_icb()*, *nrstopmag()*, *nrstartmag()*, *nrstop()*, *nrstart()*)
- *precision_mod*: This module controls the precision used in MagIC
- *logic* (*l_pressgraph()*, *l_rot_ic()*, *l_mag_nl()*, *l_dtb()*, *l_cond_ma()*, *lverbose()*, *l_mag_kin()*, *l_cond_ic()*, *l_mag_lf()*, *l_rot_ma()*, *l_mag()*): Module containing the logicals that control the run
- *riteration_mod* (riteration_t())
- *physical_parameters* (*kbotv()*, *ktopv()*): Module containing the physical parameters
- riterthetablocking_seq_mod (riterthetablocking_seq_t())
- *blocking* (*nthetabs()*, *sizethetab()*): Module containing blocking information
- *constants* (*zero()*): module containing constants and parameters used in the code.
- *truncation* (*lm_max()*, *lm_maxmag()*, *lmp_max()*, *l_maxmag()*, *l_max()*): This module defines the grid points and the truncation

Variables

Subroutines and functions

subroutine radialloop/**initialize_radialloop**()

Called from *magic*

subroutine radialloop/**finalize_radialloop**()

subroutine radialloop/**radialloopg**(*l_graph*, *l_cour*, *l_frame*, *time*, *dt*, *dtlast*, *ltocalc*, *ltonext*, *ltonext2*, *lhelcalc*, *lrmscalc*, *lviscbccalc*, *lfluxprofcalc*, *lperparcalc*, *dsdt*, *dwdt*, *dzdt*, *dpdt*, *dbdt*, *djdt*, *dvxbhlm*, *dvsrlm*, *lorentz_torque_ic*, *lorentz_torque_ma*, *br_vt_lm_cmb*, *br_vp_lm_cmb*, *br_vt_lm_icb*, *br_vp_lm_icb*, *hellmr*, *hel2lmr*, *helnalmr*, *helna2lmr*, *uhlmr*, *duhlmr*, *gradslmr*, *fconvlmr*, *fkinlmr*, *fviscslmr*, *fpoynlmr*, *freslmr*, *eperplmr*, *eparlmr*, *eperpaxilmr*, *eparaxilmr*, *dtrkc*, *dthkc*)

This subroutine performs the actual time-stepping.

Parameters

- **l_graph** *[logical,in]*
- **l_cour** *[logical,in]*
- **l_frame** *[logical,in]*
- **time** *[real,in]*

- **dt** *[real,in]*
- **dtlast** *[real,in]*
- **ltocalc** *[logical,in]*
- **ltonext** *[logical,in]*
- **ltonext2** *[logical,in]*
- **lhelcalc** *[logical,in]*
- **lrmscalc** *[logical,in]*
- **lviscbccalc** *[logical,in]*
- **lfluxprofcalc** *[logical,in]*
- **lperpparcalc** *[logical,in]*
- **dsdt** (*lm_max*,*nrstop*-(*nrstart*)+1) *[complex,out]*
- **dwdt** (*lm_max*,*nrstop*-(*nrstart*)+1) *[complex,out]*
- **dzdt** (*lm_max*,*nrstop*-(*nrstart*)+1) *[complex,out]*
- **dpdt** (*lm_max*,*nrstop*-(*nrstart*)+1) *[complex,out]*
- **dbdt** (*lm_maxmag*,*nrstopmag*-(*nrstartmag*)+1) *[complex,out]*
- **djdt** (*lm_maxmag*,*nrstopmag*-(*nrstartmag*)+1) *[complex,out]*
- **dvxbhlm** (*lm_maxmag*,*nrstopmag*-(*nrstartmag*)+1) *[complex,out]*
- **dvsrlm** (*lm_max*,*nrstop*-(*nrstart*)+1) *[complex,out]*
- **lorentz_torque_ic** *[real,out]*
- **lorentz_torque_ma** *[real,out]*
- **br_vt_lm_cmb** (*lmp_max*) *[complex,out]* :: product br*vt at CMB
- **br_vp_lm_cmb** (*lmp_max*) *[complex,out]* :: product br*vp at CMB
- **br_vt_lm_icb** (*lmp_max*) *[complex,out]* :: product br*vt at ICB
- **br_vp_lm_icb** (*lmp_max*) *[complex,out]* :: product br*vp at ICB
- **hellmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,out]*
- **hel2lmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,out]*
- **helnalmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,out]*
- **helna2lmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,out]*
- **uhlmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,out]*
- **duhlmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,out]*
- **gradslmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,out]*
- **fconvlmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,out]*
- **fkinlmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,out]*
- **fvisclmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,out]*
- **fpoynlmr** (*l_maxmag*+1,*nrstopmag*-(*nrstartmag*)+1) *[real,out]*
- **freslmr** (*l_maxmag*+1,*nrstopmag*-(*nrstartmag*)+1) *[real,out]*

- **eperplmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,out]*
- **eparlmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,out]*
- **eperpaxilmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,out]*
- **eparaxilmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,out]*
- **dtrkc** (*nrstop*-(*nrstart*)+1) *[real,out]*
- **dthkc** (*nrstop*-(*nrstart*)+1) *[real,out]*

Called from *step_time()*

Call to *graphout_mpi_header()*

10.8.2 rIteration.f90

Quick access

Routines *set_steering_variables()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC

Types

- **type** riteration_mod/**unknown_type**

  Type fields

  - **% lfluxprofcalc** *[logical]*
  - **% dthkc** *[real]*
  - **% lhelcalc** *[logical]*
  - **% lpresscalc** *[logical]*
  - **% isradialboundarypoint** *[logical]*
  - **% nbc** *[integer]*
  - **% lderiv** *[logical]*
  - **% nr** *[integer]*
  - **% l_frame** *[logical]*
  - **% l_graph** *[logical]*
  - **% lperpparcalc** *[logical]*
  - **% dtrkc** *[real]*
  - **% ltonext2** *[logical]*
  - **% l_cour** *[logical]*
  - **% ltocalc** *[logical]*
  - **% ltonext** *[logical]*

- % **lmagnlbc** *[logical]*
- % **lviscbccalc** *[logical]*
- % **lrmscalc** *[logical]*

Subroutines and functions

subroutine riteration_mod/**set_steering_variables**(*this*, *l_cour*, *ltocalc*, *ltonext*, *ltonext2*, *lderiv*, *lrmscalc*, *lhelcalc*, *l_frame*, *lmagnlbc*, *l_graph*, *lviscbccalc*, *lfluxprofcalc*, *lperpparcalc*, *lpresscalc*)

Parameters

- **this** *[real]*
- **l_cour** *[logical,in]*
- **ltocalc** *[logical,in]*
- **ltonext** *[logical,in]*
- **ltonext2** *[logical,in]*
- **lderiv** *[logical,in]*
- **lrmscalc** *[logical,in]*
- **lhelcalc** *[logical,in]*
- **l_frame** *[logical,in]*
- **lmagnlbc** *[logical,in]*
- **l_graph** *[logical,in]*
- **lviscbccalc** *[logical,in]*
- **lfluxprofcalc** *[logical,in]*
- **lperpparcalc** *[logical,in]*
- **lpresscalc** *[logical,in]*

10.8.3 rIterThetaBlocking.f90

Quick access

Routines *allocate_common_arrays()*, *deallocate_common_arrays()*, *set_thetablocking()*, *transform_to_grid_space()*, *transform_to_lm_space()*

Needed modules

- grid_space_arrays_mod(grid_space_arrays_t())
- *nonlinear_lm_mod*(nonlinear_lm_t())
- *fft*: This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5

- *legendre_spec_to_grid* (*legtfgnomag()*, *legtfg()*)
- *radial_data* (*n_r_cmb()*, *nrstart()*, *n_r_icb()*, *nrstop()*)
- *precision_mod*: This module controls the precision used in MagIC
- *legendre_grid_to_spec*
- *logic* (*l_rot_ic()*, *l_b_nl_icb()*, *l_mag_nl()*, *l_dtb()*, *l_movie_oc()*, *l_rot_ma()*, *l_cond_ic()*, *l_cond_ma()*, *l_mag_kin()*, *l_conv_nl()*, *l_b_nl_cmb()*, *l_store_frame()*, *l_heat()*, *l_mag_lf()*, *l_anel()*, *l_ht()*, *l_mag()*, *l_conv()*): Module containing the logicals that control the run
- *riteration_mod* (riteration_t())
- *physical_parameters* (*n_r_lcr()*, *kbots()*, *ktops()*): Module containing the physical parameters
- *radial_functions* (*orho1()*, *or2()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *leg_helper_mod* (leg_helper_t())
- *blocking* (*nfs()*): Module containing blocking information
- *nonlinear_bcs* (*v_rigid_boundary()*)
- *truncation* (*l_max()*, *n_r_maxstr()*, *lm_maxmag()*, *lmp_max_dtb()*, *lmp_max()*, *n_theta_maxstr()*, *lm_max()*, *nrp()*, *n_phi_maxstr()*): This module defines the grid points and the truncation

Types

- **type** riterthetablocking_mod/**unknown_type**

  Type fields

  - **% bslast** („,*) *[real,allocatable]*
  - **% bplast** („,*) *[real,allocatable]*
  - **% leg_helper** *[leg_helper_t]*
  - **% bzlast** („,*) *[real,allocatable]*
  - **% nthetabs** *[integer]*
  - **% sizethetab** *[integer]*

Subroutines and functions

subroutine riterthetablocking_mod/**allocate_common_arrays**(*this*)

Parameters **this** *[real]*

subroutine riterthetablocking_mod/**deallocate_common_arrays**(*this*)

Parameters **this** *[real]*

subroutine riterthetablocking_mod/**set_thetablocking**(*this*, *nthetabs*, *sizethetab*)

Parameters

- **this** *[real]*

- **nthetabs** *[integer,in]*
- **sizethetab** *[integer,in]*

subroutine riterthetablocking_mod/**transform_to_grid_space**(*this*, *nthetastart*, *nthetastop*, *gsa*)

Parameters

- **this** *[real]*
- **nthetastart** *[integer,in]*
- **nthetastop** *[integer,in]*
- **gsa** *[grid_space_arrays_t]*

Call to *legtfg()*, *legtfgnomag()*, *fft_thetab()*, *v_rigid_boundary()*

subroutine riterthetablocking_mod/**transform_to_lm_space**(*this*, *nthetastart*, *nthetastop*, *gsa*, *nl_lm*)

Parameters

- **this** *[real]*
- **nthetastart** *[integer,in]*
- **nthetastop** *[integer,in]*
- **gsa** *[grid_space_arrays_t]*
- **nl_lm** *[nonlinear_lm_t]*

Call to *fft_thetab()*, *legtf3()*, *legtf2()*, *legtf1()*

10.8.4 rIterThetaBlocking_OpenMP.f90

Quick access

Routines *finalize_riterthetablocking_openmp()*, *initialize_riterthetablocking_openmp()*, *do_iteration_thetablocking_openmp()*

Needed modules

- grid_space_arrays_mod(grid_space_arrays_t())
- *dtb_arrays_mod*(dtb_arrays_t())
- *outrot*(*get_lorentz_torque()*)
- *riterthetablocking_mod*(riterthetablocking_t())
- *graphout_mod*(*graphout_mpi()*)
- *out_movie*(*store_movie_frame()*)
- *nonlinear_lm_mod*(nonlinear_lm_t())
- *torsional_oscillations*(*gettonext()*, *getto()*, *gettofinish()*): This module contains information for TO calculation and output
- *radial_data*(*n_r_cmb()*, *n_r_icb()*)
- *precision_mod*: This module controls the precision used in MagIC

- *dtb_mod* (*get_dh_dtblm()*, *get_dtblm()*): This module contains magnetic field stretching and advection terms plus a separate omega-effect. It is used for movie output....
- *to_arrays_mod* (to_arrays_t())
- *nonlinear_bcs* (*get_br_v_bcs()*)
- *logic* (*l_rot_ic()*, *l_b_nl_icb()*, *l_mag_nl()*, *l_dtb()*, *l_movie_oc()*, *l_rot_ma()*, *l_cond_ic()*, *l_cond_ma()*, *l_mag_kin()*, *l_conv_nl()*, *l_b_nl_cmb()*, *l_store_frame()*, *l_heat()*, *l_mag_lf()*, *l_anel()*, *l_to()*, *l_ht()*, *l_mag()*, *l_conv()*): Module containing the logicals that control the run
- *courant_mod* (*courant()*)
- *nl_special_calc*: This module allows to calculcate several diagnostics that need to be computed in the physical space (non-linear quantities)
- *radial_functions* (*orho1()*, *or2()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *leg_helper_mod* (leg_helper_t())
- *blocking* (*nfs()*): Module containing blocking information
- *constants* (*zero()*): module containing constants and parameters used in the code.
- *truncation* (*l_max()*, *n_r_maxstr()*, *lmp_max_dtb()*, *lmp_max()*, *n_theta_maxstr()*, *lm_max()*, *nrp()*, *n_phi_maxstr()*): This module defines the grid points and the truncation

Types

- **type** riterthetablocking_openmp_mod/**unknown_type**

  Type fields

  - **% gsa** (*) *[grid_space_arrays_t,allocatable]*
  - **% lorentz_torque_ic** (*) *[real,allocatable]*
  - **% nthreads** *[integer]*
  - **% dtb_arrays** (*) *[dtb_arrays_t,allocatable]*
  - **% nl_lm** (*) *[nonlinear_lm_t,allocatable]*
  - **% to_arrays** (*) *[to_arrays_t,allocatable]*
  - **% lorentz_torque_ma** (*) *[real,allocatable]*

Subroutines and functions

function riterthetablocking_openmp_mod/**getthistype**(*this*)

Parameters **this** *[real]*

Return **getthistype** *[character]*

subroutine riterthetablocking_openmp_mod/**initialize_riterthetablocking_openmp**(*this*)

Parameters **this** *[real]*

subroutine riterthetablocking_openmp_mod/**finalize_riterthetablocking_openmp**(*this*)

Parameters **this** *[real]*

subroutine riterthetablocking_openmp_mod/**do_iteration_thetablocking_openmp**(*this*, *nr*, *nbc*, *time*, *dt*, *dt-last*, *dsdt*, *dwdt*, *dzdt*, *dpdt*, *dbdt*, *djdt*, *dvxbhlm*, *dvs-rlm*, *br_vt_lm_cmb*, *br_vp_lm_cmb*, *br_vt_lm_icb*, *br_vp_lm_icb*, *lorentz_torque_ic*, *lorentz_torque_ma*, *hellmr*, *hel2lmr*, *hel-nalmr*, *helna2lmr*, *uhlmr*, *duhlmr*, *grad-slmr*, *fcon-vlmr*, *fkinlmr*, *fvis-clmr*, *fpoynlmr*, *freslmr*, *eper-plmr*, *eparlmr*, *eper-pax-ilmr*, *eparax-ilmr*)

Parameters

- **this** *[real]*
- **nr** *[integer,in]*
- **nbc** *[integer,in]*
- **time** *[real,in]*

- **dt** *[real,in]*
- **dtlast** *[real,in]*
- **dsdt** (*) *[complex,out]*
- **dwdt** (*) *[complex,out]*
- **dzdt** (*) *[complex,out]*
- **dpdt** (*) *[complex,out]*
- **dbdt** (*) *[complex,out]*
- **djdt** (*) *[complex,out]*
- **dvxbhlm** (*) *[complex,out]*
- **dvsrlm** (*) *[complex,out]*
- **br_vt_lm_cmb** (*) *[complex,out]*
- **br_vp_lm_cmb** (*) *[complex,out]*
- **br_vt_lm_icb** (*) *[complex,out]*
- **br_vp_lm_icb** (*) *[complex,out]*
- **lorentz_torque_ic** *[real,out]*
- **lorentz_torque_ma** *[real,out]*
- **hellmr** (*) *[real,out]*
- **hel2lmr** (*) *[real,out]*
- **helnalmr** (*) *[real,out]*
- **helna2lmr** (*) *[real,out]*
- **uhlmr** (*) *[real,out]*
- **duhlmr** (*) *[real,out]*
- **gradslmr** (*) *[real,out]*
- **fconvlmr** (*) *[real,out]*
- **fkinlmr** (*) *[real,out]*
- **fvisclmr** (*) *[real,out]*
- **fpoynlmr** (*) *[real,out]*
- **freslmr** (*) *[real,out]*
- **eperplmr** (*) *[real,out]*
- **eparlmr** (*) *[real,out]*
- **eperpaxilmr** (*) *[real,out]*
- **eparaxilmr** (*) *[real,out]*

Call to *get_br_v_bcs()*, *get_lorentz_torque()*, *courant()*, *graphout_mpi()*, *get_helicity()*, *get_nlblayers()*, *get_fluxes()*, *get_perppar()*, *store_movie_frame()*, *get_dtblm()*, *gettonext()*, *getto()*, *gettofinish()*, *get_dh_dtblm()*

10.8.5 get_nl.f90

Types

- **type** general_arrays_mod/**unknown_type**

10.8.6 get_td.f90

Quick access

Routines *finalize()*, *set_zero()*, *get_td()*

Needed modules

- *horizontal_data* (*dtheta3a()*, *hdif_b()*, *dphi0()*, *dtheta4a()*, *dtheta1a()*, *dlh()*, *dtheta2a()*, *dtheta3s()*, *hdif_v()*, *dtheta4s()*, *dtheta1s()*, *dphi()*, *dtheta2s()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *rms* (*lf2hint()*, *pre2hint()*, *clf2hint()*, *geo2hint()*, *arc2hint()*, *cia2hint()*, *cor2hint()*, *adv2hint()*, *mag2hint()*, *buo2hint()*, *plf2hint()*): This module contains the calculation of thr RMS force balance and induction terms.
- *fields* (*w_rloc()*, *z_rloc()*, *dw_rloc()*): This module contains the potential fields and their radial derivatives
- *rms_helpers* (*hintrms()*): This module contains several useful subroutines required to compute RMS diagnostics
- *precision_mod*: This module controls the precision used in MagIC
- *logic* (*l_anelastic_liquid()*, *l_mag_nl()*, *l_mag_kin()*, *l_conv_nl()*, *l_mag_lf()*, *l_anel()*, *l_heat()*, *l_rms()*, *l_conv()*, *l_mag()*, *l_corr()*): Module containing the logicals that control the run
- *physical_parameters* (*epsc()*, *vischeatfac()*, *ra()*, *n_r_lcr()*, *corfac()*, *ohmlossfac()*): Module containing the physical parameters
- *truncation* (*lm_max()*, *l_max()*, *lmp_max()*, *lm_maxmag()*): This module defines the grid points and the truncation
- *leg_helper_mod* (leg_helper_t())
- *blocking* (*lmp2lmps()*, *lm2lms()*, *lm2lmp()*, *lm2m()*, *lm2l()*, *lmp2lmpa()*, *st_map()*, *lm2lma()*): Module containing blocking information
- *constants* (*zero()*, *two()*): module containing constants and parameters used in the code.
- *radial_functions* (*epscprof()*, *or4()*, *rho0()*, *or2()*, *or1()*, *beta()*, *r()*, *temp0()*, *rgrav()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Types

- **type** nonlinear_lm_mod/**unknown_type**

  Type fields

- **% vxbrlm** (*) *[complex,allocatable]*
- **% vstlm** (*) *[complex,allocatable]*
- **% p1lm** (*) *[complex,allocatable]*
- **% vsplm** (*) *[complex,allocatable]*
- **% lft2lm** (*) *[complex,allocatable]*
- **% lfplm** (*) *[complex,allocatable]*
- **% lfrlm** (*) *[complex,allocatable]*
- **% lftlm** (*) *[complex,allocatable]*
- **% advtlm** (*) *[complex,allocatable]*
- **% advrlm** (*) *[complex,allocatable]*
- **% advplm** (*) *[complex,allocatable]*
- **% vxbtlm** (*) *[complex,allocatable]*
- **% cfp2lm** (*) *[complex,allocatable]*
- **% advt2lm** (*) *[complex,allocatable]*
- **% vsrlm** (*) *[complex,allocatable]*
- **% lfp2lm** (*) *[complex,allocatable]*
- **% vxbplm** (*) *[complex,allocatable]*
- **% p2lm** (*) *[complex,allocatable]*
- **% ohmlosslm** (*) *[complex,allocatable]*
- **% vischeatlm** (*) *[complex,allocatable]*
- **% cft2lm** (*) *[complex,allocatable]*
- **% advp2lm** (*) *[complex,allocatable]*

Subroutines and functions

subroutine nonlinear_lm_mod/**initialize**(*this*, *lmp_max*)

Parameters

- **this** *[real]*
- **lmp_max** *[integer,in]*

subroutine nonlinear_lm_mod/**finalize**(*this*)

Parameters **this** *[real]*

subroutine nonlinear_lm_mod/**set_zero**(*this*)

Parameters **this** *[real]*

subroutine nonlinear_lm_mod/**output**(*this*)

Parameters **this** *[real]*

subroutine nonlinear_lm_mod/**get_td**(*this*, *nr*, *nbc*, *lrmscalc*, *dvsrlm*, *dvxbhlm*, *dwdt*, *dzdt*, *dpdt*, *dsdt*, *dbdt*, *djdt*, *leg_helper*)

Purpose of this to calculate time derivatives dwdt,dzdt,dpdt,dsdt,dbdt,djdt and auxiliary arrays dVSrLM and dVxBhLM from non-linear terms in spectral form, contained in flmw1-3,flms1-3, flmb1-3 (input)

Parameters

- **this** *[real]*
- **nr** *[integer,in]*
- **nbc** *[integer,in]* :: signifies boundary conditions
- **lrmscalc** *[logical,in]*
- **dvsrlm** (*lm_max*) *[complex,out]*
- **dvxbhlm** (*lm_maxmag*) *[complex,out]*
- **dwdt** (*lm_max*) *[complex,out]*
- **dzdt** (*lm_max*) *[complex,out]*
- **dpdt** (*lm_max*) *[complex,out]*
- **dsdt** (*lm_max*) *[complex,out]*
- **dbdt** (*lm_maxmag*) *[complex,out]*
- **djdt** (*lm_maxmag*) *[complex,out]*
- **leg_helper** *[leg_helper_t,in]*

Call to *hintrms()*

10.8.7 nonlinear_bcs.f90

Quick access

Routines *get_b_nl_bcs()*, *get_br_v_bcs()*, *v_rigid_boundary()*

Needed modules

- *horizontal_data* (*dtheta1a()*, *dlh()*, *sn2()*, *o_sin_theta()*, *dtheta1s()*, *dphi()*, *costheta()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *legendre_grid_to_spec* (*legtf2()*)
- *fft* (*fft_thetab()*): This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5
- *radial_data* (*n_r_cmb()*, *n_r_icb()*)
- *precision_mod*: This module controls the precision used in MagIC
- *physical_parameters* (*prmag()*, *conductance_ma()*, *sigma_ratio()*): Module containing the physical parameters
- *truncation* (*nrp()*, *n_phi_max()*, *lmp_max()*): This module defines the grid points and the truncation
- *blocking* (*lmp2lmps()*, *lm2lmp()*, *sizethetab()*, *lm2m()*, *lm2l()*, *nfs()*, *lmp2lmpa()*): Module containing blocking information

- *constants* (*two()*): module containing constants and parameters used in the code.
- *radial_functions* (*r_cmb()*, *r_icb()*, *rho0()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

Subroutines and functions

subroutine nonlinear_bcs/**get_br_v_bcs**(*br*, *vt*, *vp*, *omega*, *o_r_e_2*, *o_rho*, *n_theta_min*, *n_theta_block*, *br_vt_lm*, *br_vp_lm*)

Purpose of this subroutine is to calculate the nonlinear term of the magnetic boundary condition for a conducting mantle or inner core in space (r,lm). Calculation is performed for the theta block:

```
n_theta_min<=n_theta<=n_theta_min+n_theta_block-1
```

On input br, vt and vp are given on all phi points and thetas in the specific block. On output the contribution of these grid points to all degree and orders is stored in br_vt_lm and br_vp_lm. Output is [r/sin(theta)*Br*U]=[(0,br_vt_lm,br_vp_lm)]

Parameters

- **br** (*nrp*,*) *[real,in]* :: r**2 * B_r
- **vt** (*nrp*,*) *[real,in]* :: r*sin(theta) U_theta
- **vp** (*nrp*,*) *[real,in]* :: r*sin(theta) U_phi
- **omega** *[real,in]* :: rotation rate of mantle or IC
- **o_r_e_2** *[real,in]* :: 1/r**2
- **o_rho** *[real,in]* :: 1/rho0 (anelastic)
- **n_theta_min** *[integer,in]* :: start of theta block
- **n_theta_block** *[integer,in]* :: size of theta_block
- **br_vt_lm** (*lmp_max*) *[complex,inout]*
- **br_vp_lm** (*lmp_max*) *[complex,inout]*

Called from *do_iteration_thetablocking_openmp()*, do_iteration_thetablocking_seq(), do_iteration_thetablocking_shtns()

Call to *fft_thetab()*, *legtf2()*

subroutine nonlinear_bcs/**get_b_nl_bcs**(*bc*, *br_vt_lm*, *br_vp_lm*, *lm_min_b*, *lm_max_b*, *b_nl_bc*, *aj_nl_bc*)

Parameters

- **bc** *[character,in]* :: Distinguishes 'CMB' and 'ICB'
- **br_vt_lm** (*lmp_max*) *[complex,in]* :: [br*vt/(r**2*sin(theta)**2)]
- **br_vp_lm** (*lmp_max*) *[complex,in]* :: [br*vp/(r**2*sin(theta)**2)
- **lm_min_b** *[integer,in]* :: limits of lm-block
- **lm_max_b** *[integer,in]* :: nonlinear bc for aj
- **b_nl_bc** (lm_max_b-lm_min_b+1) *[complex,out]*

- **aj_nl_bc** (lm_max_b-lm_min_b+1) *[complex,out]*

Called from *step_time()*

subroutine nonlinear_bcs/**v_rigid_boundary** (*nr*, *omega*, *lderiv*, *vrr*, *vtr*, *vpr*, *cvrr*, *dvrdtr*, *dvrdpr*, *dvtdpr*, *dvpdpr*, *nthetastart*)

Purpose of this subroutine is to set the velocities and their derivatives at a fixed boundary. While vt is zero, since we only allow for rotation about the z-axix, vp= r sin(theta) v_phi = r**2 sin(theta)**2 omega cvr= r**2 * radial component of (curl v) = r**2 2 cos(theta) omega

Parameters

- **nr** *[integer,in]* :: no of radial grid point
- **omega** *[real,in]*
- **lderiv** *[logical,in]* :: derivatives required ?
- **vrr** (*nrp*,*nfs*) *[real,out]*
- **vtr** (*nrp*,*nfs*) *[real,out]*
- **vpr** (*nrp*,*nfs*) *[real,out]*
- **cvrr** (*nrp*,*nfs*) *[real,out]*
- **dvrdtr** (*nrp*,*nfs*) *[real,out]*
- **dvrdpr** (*nrp*,*nfs*) *[real,out]*
- **dvtdpr** (*nrp*,*nfs*) *[real,out]*
- **dvpdpr** (*nrp*,*nfs*) *[real,out]*
- **nthetastart** *[integer,in]* :: no of theta to start with

Called from transform_to_grid_space_shtns(), *transform_to_grid_space()*

10.9 Chebyshev polynomials and cosine transforms

10.9.1 chebyshev_polynoms.f90

Quick access

Variables *get_chebs*

Routines *cheb_grid()*, *get_chebs_direct()*, *get_chebs_recurr()*, *get_chebs_even()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *constants* (*one()*, *four()*, *pi()*, *two()*, *half()*): module containing constants and parameters used in the code.
- *logic* (*l_newmap()*): Module containing the logicals that control the run

Variables

- chebyshev_polynoms_mod/**get_chebs** *[public]*

Subroutines and functions

subroutine chebyshev_polynoms_mod/**get_chebs_recurr**(*n_r*, *a*, *b*, *y*, *n_r_max*, *cheb*, *dcheb*, *d2cheb*, *d3cheb*, *dim1*, *dim2*, *map_fac1*, *map_fac2*, *map_fac3*)

Construct Chebychev polynomials and their first, second, and third derivative up to degree n_r at n_r points x in the interval [a,b]. Since the Chebs are only defined in [-1,1] we have to use a map, mapping the points x points y in the interval [-1,1]. This map is executed by the subroutine cheb_grid and has to be done before calling this program.

Parameters

- **n_r** *[integer,in]* :: number of grid points
- **a** *[real,in]* :: interval boundaries [a,b]
- **b** *[real,in]*
- **y** (*n_r_max*) *[real,in]* :: n_r grid points in interval [a,b]
- **n_r_max** *[integer,in,]* :: leading dimension of
- **cheb** (dim1,dim2) *[real,out]* :: cheb(i,j) is Chebychev pol.
- **dcheb** (dim1,dim2) *[real,out]* :: first derivative of cheb
- **d2cheb** (dim1,dim2) *[real,out]* :: second derivative o cheb
- **d3cheb** (dim1,dim2) *[real,out]* :: third derivative of cheb
- **dim1** *[integer,in]* :: dimensions of cheb,dcheb,....
- **dim2** *[integer,in]*
- **map_fac1** (*n_r_max*) *[real,in]*
- **map_fac2** (*n_r_max*) *[real,in]*
- **map_fac3** (*n_r_max*) *[real,in]*

subroutine chebyshev_polynoms_mod/**get_chebs_even**(*n_r*, *a*, *b*, *y*, *n_r_max*, *cheb*, *dcheb*, *d2cheb*, *dim1*, *dim2*)

Parameters

- **n_r** *[integer,in]* :: number of grid points
- **a** *[real,in]* :: interval boundaries [a,b]
- **b** *[real,in]*
- **y** (*n_r_max*) *[real,in]* :: n_r grid points in interval [a,b]
- **n_r_max** *[integer,in,]* :: max number of radial points, dims of y
- **cheb** (dim1,dim2) *[real,out]* :: cheb(i,j) is Chebychev pol.
- **dcheb** (dim1,dim2) *[real,out]* :: first derivative of cheb
- **d2cheb** (dim1,dim2) *[real,out]* :: second derivative o cheb

- **dim1** *[integer,in]* :: dimensions of cheb,dcheb,.....
- **dim2** *[integer,in]*

Called from *radial()*

subroutine chebyshev_polynoms_mod/**get_chebs_direct**(*n_r*, *a*, *b*, *y*, *n_r_max*, *cheb*, *dcheb*, *d2cheb*, *d3cheb*, *dim1*, *dim2*, *map_fac1*, *map_fac2*, *map_fac3*)

Construct Chebychev polynomials and their first, second, and third derivative up to degree n_r at n_r points x in the interval [a,b]. Since the Chebs are only defined in [-1,1] we have to use a map, mapping the points x points y in the interval [-1,1]. This map is executed by the subroutine cheb_grid and has to be done before calling this program.

Parameters

- **n_r** *[integer,in]* :: number of grid points
- **a** *[real,in]* :: interval boundaries [a,b]
- **b** *[real,in]*
- **y** (*n_r_max*) *[real,in]* :: n_r grid points in interval [a,b]
- **n_r_max** *[integer,in,]* :: leading dimension of
- **cheb** (dim1,dim2) *[real,out]* :: cheb(i,j) is Chebychev pol.
- **dcheb** (dim1,dim2) *[real,out]* :: first derivative of cheb
- **d2cheb** (dim1,dim2) *[real,out]* :: second derivative o cheb
- **d3cheb** (dim1,dim2) *[real,out]* :: third derivative of cheb
- **dim1** *[integer,in]* :: dimensions of cheb,dcheb,....
- **dim2** *[integer,in]*
- **map_fac1** (*n_r_max*) *[real,in]*
- **map_fac2** (*n_r_max*) *[real,in]*
- **map_fac3** (*n_r_max*) *[real,in]*

subroutine chebyshev_polynoms_mod/**cheb_grid**(*a*, *b*, *n*, *x*, *y*, *a1*, *a2*, *x0*, *lbd*)

Given the interval [a,b] the routine returns the n+1 points that should be used to support a Chebychev expansion. These are the n+1 extrema y(i) of the Chebychev polynomial of degree n in the interval [-1,1]. The respective points mapped into the interval of question [a,b] are the x(i).

Note: x(i) and y(i) are stored in the reversed order: x(1)=b, x(n+1)=a, y(1)=1, y(n+1)=-1

Parameters

- **a** *[real,in]* :: interval boundaries
- **b** *[real,in]*
- **n** *[integer,in]* :: degree of Cheb polynomial to be represented by the grid points
- **x** (*) *[real,out]* :: grid points in interval [a,b]
- **y** (*) *[real,out]* :: grid points in interval [-1,1]
- **a1** *[real,in]*

- **a2** *[real,in]*
- **x0** *[real,in]*
- **lbd** *[real,in]*

Called from *chebintinit()*, *init_rnb()*, *radial()*

10.9.2 cosine_transform_odd.f90

Quick access

Routines *costf1_complex_1d()*, *costf1_real_1d()*, *costf1_complex()*, *costf1_real()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *fft_fac_mod* (*fft_fac_complex()*, *fft_fac_real()*)
- *useful* (*factorise()*): library with several useful subroutines
- *constants* (*cos36()*, *sin36()*, *cos72()*, *two()*, *one()*, *sin72()*, *sin60()*, *half()*, *pi()*): module containing constants and parameters used in the code.
- *truncation* (*lm_max()*, *lm_max_real()*): This module defines the grid points and the truncation

Types

- **type** cosine_transform_odd/**unknown_type**

  Type fields

  - **% i_costf_init** (*) *[integer,allocatable]*
  - **% d_costf_init** (*) *[real,allocatable]*

Subroutines and functions

subroutine cosine_transform_odd/**initialize**(*this*, *n*, *ni*, *nd*)

Purpose of this subroutine is to calculate and store several values that will be needed for a fast cosine transform of the first kind. The actual transform is performed by the subroutine costf1.

Parameters

- **this** *[real]*
- **n** *[integer,in]*
- **ni** *[integer,in]*
- **nd** *[integer,in]*

Call to *factorise()*

subroutine cosine_transform_odd/**finalize**(*this*)

Parameters **this** *[real]*

subroutine cosine_transform_odd/**costf1_complex**(*this*, *f*, *n_f_max*, *n_f_start*, *n_f_stop*, *f2*)

Purpose of this subroutine is to perform a multiple cosine transforms for n+1 datapoints on the columns numbered n_f_start to n_f_stop in the array f(n_f_max,n+1) Depending whether the input f contains data or coeff arrays coeffs or data are returned in f.

Parameters

- **this** *[real]*
- **f** (n_f_max,*) *[complex,inout]* :: data/coeff input
- **n_f_max** *[integer,in,]* :: number of columns in f,f2
- **n_f_start** *[integer,in]* :: columns to be transformed
- **n_f_stop** *[integer,in]*
- **f2** (n_f_max,*) *[complex,out]* :: work array of the same size as f

Call to *fft_fac_complex()*

subroutine cosine_transform_odd/**costf1_complex_1d**(*this*, *f*, *f2*)

Parameters

- **this** *[real]*
- **f** (*) *[complex,inout]* :: data/coeff input
- **f2** (*) *[complex,out]* :: work array of the same size as f

Call to *fft_fac_complex()*

subroutine cosine_transform_odd/**costf1_real**(*this*, *f*, *n_f_max*, *n_f_start*, *n_f_stop*, *f2*)

Parameters

- **this** *[real]*
- **f** (n_f_max,*) *[real,inout]* :: data/coeff input
- **n_f_max** *[integer,in,]* :: number of columns in f,f2
- **n_f_start** *[integer,in]* :: columns to be transformed
- **n_f_stop** *[integer,in]*
- **f2** (n_f_max,*) *[real,out]* :: work array of the same size as f

Call to *fft_fac_real()*

subroutine cosine_transform_odd/**costf1_real_1d**(*this*, *f*, *f2*)

Parameters

- **this** *[real]*
- **f** (*) *[real,inout]* :: data/coeff input
- **f2** (*) *[real,out]* :: work array

Call to *fft_fac_real()*

10.9.3 cosine_transform_even.f90

Quick access

Routines *costf2()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *fft_fac_mod* (*fft_fac_complex()*)
- *useful* (*factorise()*): library with several useful subroutines
- *constants* (*cos36()*, *sin36()*, *cos72()*, *two()*, *one()*, *sin72()*, *sin60()*, *half()*, *pi()*): module containing constants and parameters used in the code.
- *truncation* (*lm_max()*): This module defines the grid points and the truncation

Types

- **type** cosine_transform_even/**unknown_type**

  Type fields

  - **% i_costf_init** (*) *[integer,allocatable]*
  - **% d_costf_init** (*) *[real,allocatable]*

Subroutines and functions

subroutine cosine_transform_even/**initialize**(*this*, *n*, *ni*, *nd*)

Purpose of this subroutine is to calculate several things needed for the cheb transform. Prepares costf2 for even number of grid points.

Parameters

- **this** *[real]*
- **n** *[integer,in]*
- **ni** *[integer,in]*
- **nd** *[integer,in]*

Call to *factorise()*

subroutine cosine_transform_even/**finalize**(*this*)

Parameters **this** *[real]*

subroutine cosine_transform_even/**costf2**(*this*, *f*, *n_f_max*, *n_f_start*, *n_f_stop*, *f2*, *isign*)

Purpose of this subroutine is to perform a multiple cosine transforms for n+1 datapoints on the columns numbered n_f_start to n_f_stop in the array y(n_f_max,n+1) Depending whether the input y contains data or coeff arrays coeffs or data are returned in y.

Parameters

- **this** *[real]*
- **f** (n_f_max,*) *[complex,inout]* :: data/coeff input
- **n_f_max** *[integer,in,]* :: number of columns in y,y2
- **n_f_start** *[integer,in]* :: columns to be transformed
- **n_f_stop** *[integer,in]*
- **f2** (n_f_max,*) *[complex,out]* :: work array of the same size as y
- **isign** *[integer,in]* :: = +1 (-1) for forward (backward) transform

Call to *fft_fac_complex()*

10.9.4 fft_fac.f90

Quick access

Routines *fft_fac_complex()*, *fft_fac_real()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *constants* (*cos36()*, *sin36()*, *cos72()*, *sin72()*, *sin60()*, *half()*): module containing constants and parameters used in the code.

Variables

Subroutines and functions

subroutine fft_fac_mod/**fft_fac_real** (*a*, *b*, *c*, *d*, *trigs*, *nv*, *l1*, *l2*, *n*, *ifac*, *la*)

main part of Fourier / Chebychev transform called in costf1, costf2

Parameters

- **a** (*) *[real,in]*
- **b** (*) *[real,in]*
- **c** (*) *[real,out]*
- **d** (*) *[real,out]*
- **trigs** (2 * n) *[real,in]*
- **nv** *[integer,in]*
- **l1** *[integer,in]*
- **l2** *[integer,in]*
- **n** *[integer,in,]*
- **ifac** *[integer,in]*
- **la** *[integer,in]*

Called from *costf1_real_1d()*, *costf1_real()*

subroutine fft_fac_mod/**fft_fac_complex**(*a*, *b*, *c*, *d*, *trigs*, *nv*, *l1*, *l2*, *n*, *ifac*, *la*)

main part of Fourier / Chebychev transform called in costf1, costf2

Parameters

- **a** (*) *[complex,in]*
- **b** (*) *[complex,in]*
- **c** (*) *[complex,out]*
- **d** (*) *[complex,out]*
- **trigs** (2 * n) *[real,in]*
- **nv** *[integer,in]*
- **l1** *[integer,in]*
- **l2** *[integer,in]*
- **n** *[integer,in,]*
- **ifac** *[integer,in]*
- **la** *[integer,in]*

Called from *costf1_complex_1d()*, *costf2()*, *costf1_complex()*

10.10 Legendre transforms

10.10.1 plms.f90

Quick access

Routines *plm_theta()*, *plm_thetaas()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *constants* (*osq4pi()*, *two()*, *one()*): module containing constants and parameters used in the code.

Variables

Subroutines and functions

subroutine plms_theta/**plm_theta**(*theta*, *max_degree*, *max_order*, *m0*, *plma*, *dtheta_plma*, *ndim_plma*, *norm*)

Parameters

- **theta** *[real,in]* :: angle in degrees
- **max_degree** *[integer,in]* :: required max degree of plm

- **max_order** *[integer,in]* :: required max order of plm
- **m0** *[integer,in]* :: basic wave number
- **plma** (ndim_plma) *[real,out]* :: ass. legendres at theta
- **dtheta_plma** (ndim_plma) *[real,out]* :: their theta derivative
- **ndim_plma** *[integer,in]* :: dimension of plma and dtheta_plma
- **norm** *[integer,in]* :: =0 fully normalised

Called from *outpv()*, *getegeos()*, *lnpas2tr()*, *horizontal()*, *outto()*

subroutine plms_theta/**plm_thetaas**(*theta*, *max_degree*, *plma*, *dtheta_plma*, *ndim_plma*, *norm*)

The produces the plm for all degrees and order=0 for a given theta plus dtheta_plma=sin(theta)* (d plm)/(d theta)

Norm determins the normalisation: n=0 – surface normalised, n=1 – Schmidt normalised, n=2 – fully normalised.

Parameters

- **theta** *[real,in]* :: angle in degrees
- **max_degree** *[integer,in]* :: required max degree of plm
- **plma** (ndim_plma) *[real,out]* :: ass. legendres at theta
- **dtheta_plma** (ndim_plma) *[real,out]* :: their theta derivative
- **ndim_plma** *[integer,in]* :: dimension of plma and dtheta_plma
- **norm** *[integer,in]* :: =0 fully normalised

Called from *horizontal()*

10.10.2 legendre_helpers.f90

Quick access

Routines *initialize()*, *legprepg()*, *legprep()*, *legprep_ic()*

Needed modules

- *horizontal_data* (*dlh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *torsional_oscillations* (*ddzasl()*): This module contains information for TO calculation and output
- *fields* (*z_rloc()*, *s_rloc()*, *ddw_rloc()*, *dj_rloc()*, *b_rloc()*, *ddb_rloc()*, *db_rloc()*, *dw_rloc()*, *w_rloc()*, *p_rloc()*, *aj_rloc()*, *dp_rloc()*, *dz_rloc()*, *ds_rloc()*, *omega_ma()*, *omega_ic()*): This module contains the potential fields and their radial derivatives
- *radial_data* (*n_r_cmb()*, *n_r_icb()*)
- *precision_mod*: This module controls the precision used in MagIC
- *logic* (*l_movie_oc()*, *l_mag_kin()*, *l_heat()*, *l_mag_lf()*, *l_fluxprofs()*, *l_mag()*, *l_conv()*): Module containing the logicals that control the run

- *truncation* (*lm_max()*, *l_max()*, *n_m_max()*): This module defines the grid points and the truncation
- *grenoble* (*lgrenoble()*, *ddb0()*, *b0()*, *db0()*): This module contains all variables for the case of an imposed IC dipole
- *blocking* (*lm2m()*, *lm2l()*, *lm2()*): Module containing blocking information
- *constants* (*zero()*, *two()*, *one()*): module containing constants and parameters used in the code.
- *radial_functions* (*or2()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Types

- **type** leg_helper_mod/**unknown_type**

  Type fields

  - **% bcmb** (*) *[complex,allocatable]*
  - **% omegaic** *[real]*
  - **% dlhz** (*) *[complex,allocatable]*
  - **% bhc** (*) *[complex,allocatable]*
  - **% dlhw** (*) *[complex,allocatable]*
  - **% bhg** (*) *[complex,allocatable]*
  - **% omegama** *[real]*
  - **% dlhj** (*) *[complex,allocatable]*
  - **% dlhb** (*) *[complex,allocatable]*
  - **% zas** (*) *[real,allocatable]*
  - **% dzas** (*) *[real,allocatable]*
  - **% ddzas** (*) *[real,allocatable]*
  - **% dpr** (*) *[complex,allocatable]*
  - **% dlhdw** (*) *[complex,allocatable]*
  - **% sr** (*) *[complex,allocatable]*
  - **% dsr** (*) *[complex,allocatable]*
  - **% dvhdrg** (*) *[complex,allocatable]*
  - **% dvhdrc** (*) *[complex,allocatable]*
  - **% cbhg** (*) *[complex,allocatable]*
  - **% vhg** (*) *[complex,allocatable]*
  - **% cbhc** (*) *[complex,allocatable]*
  - **% vhc** (*) *[complex,allocatable]*
  - **% prer** (*) *[complex,allocatable]*

Variables

Subroutines and functions

subroutine leg_helper_mod/**initialize**(*this*, *lm_max*, *lm_maxmag*, *l_max*)

Parameters

- **this** *[real]*
- **lm_max** *[integer,in]*
- **lm_maxmag** *[integer,in]*
- **l_max** *[integer,in]*

subroutine leg_helper_mod/**legprepg**(*this*, *nr*, *nbc*, *lderiv*, *lrmscalc*, *lpresscalc*, *l_frame*, *ltonext*, *ltonext2*, *ltocalc*)

Purpose of this subroutine is to prepare Legendre transforms from (r,l,m) space to (r,theta,m) space by calculating auxiliary arrays dpdw,dpddw, ... dLhj which contain combinations of harmonic coeffs multiplied with (l,m)-dependend factors as well as the radial dependence:

•nBc =0 standard inner radial grid point

•nBc =1 radial velocity zero, spatial derivs not needed

•nBc =2 all velocity comp. zero, spatial derivs not needed

lDeriv=.true. field derivatives required

Parameters

- **this** *[real]*
- **nr** *[integer,in]* :: radial level
- **nbc** *[integer,in]* :: boundary condition
- **lderiv** *[logical,in]* :: get also field derivatives !
- **lrmscalc** *[logical,in]* :: Rms force balance ?
- **lpresscalc** *[logical,in]* :: Pressure ?
- **l_frame** *[logical,in]* :: Calculate movie frame?
- **ltonext** *[logical,in]* :: for TO output
- **ltonext2** *[logical,in]*
- **ltocalc** *[logical,in]*

subroutine leg_helper_mod/**legprep**(*w*, *dw*, *ddw*, *z*, *dz*, *dllh*, *lm_max*, *l_max*, *minc*, *r*, *lderiv*, *lhor*, *dllhw*, *vhg*, *vhc*, *dllhz*, *cvhg*, *cvhc*)

Purpose of this subroutine is to prepare Legendre transforms from (r,l,m) space to (r,theta,m) space by calculating auxiliary arrays w, dw, ddw, ... which contain combinations of harmonic coeffs multiplied with (l,m)-dependend factors as well as possible the radial dependencies.

lDeriv=.true. field derivatives required for curl of field

Parameters

- **w** (*lm_max*) *[complex,in]*

- **dw** (*lm_max*) *[complex,in]*
- **ddw** (*lm_max*) *[complex,in]*
- **z** (*lm_max*) *[complex,in]*
- **dz** (*lm_max*) *[complex,in]*
- **dlh** (*lm_max*) *[real,in]*
- **lm_max** *[integer,in,]*
- **l_max** *[integer,in]*
- **minc** *[integer,in]*
- **r** *[real,in]*
- **lderiv** *[logical,in]*
- **lhor** *[logical,in]*
- **dlhw** (*) *[complex,out]*
- **vhg** (*) *[complex,out]*
- **vhc** (*) *[complex,out]*
- **dlhz** (*) *[complex,out]*
- **cvhg** (*) *[complex,out]*
- **cvhc** (*) *[complex,out]*

Called from *fields_average()*

subroutine leg_helper_mod/**legprep_ic**(*w*, *dw*, *ddw*, *z*, *dz*, *dlh*, *lm_max*, *l_max*, *minc*, *r*, *r_icb*, *lderiv*, *lhor*, *lcondic*, *dlhw*, *vhg*, *vhc*, *dlhz*, *cvhg*, *cvhc*)

Purpose of this subroutine is to prepare Legendre transforms from (r,l,m) space to (r,theta,m) space by calculating auxiliary arrays dLhw,vhG, ... which contain combinations of harmonic coeffs multiplied with (l,m)-dependend factors as well as possible the radial dependencies.

lDeriv=.true. field derivatives required for curl of field

Note: This routine is used for the inner core magnetic field which has a special radial function ansatz. It can also be used to prepare the calculation of a field in an insulating inner core for lCondIC=.false.. For this the w has to be the outer core poloidal field and nR is the grid point for the ICB. In any case legTF can be used for the following Legendre transform and fftJW for the Fourier transform.

Parameters

- **w** (*lm_max*) *[complex,in]*
- **dw** (*lm_max*) *[complex,in]*
- **ddw** (*lm_max*) *[complex,in]*
- **z** (*lm_max*) *[complex,in]*
- **dz** (*lm_max*) *[complex,in]*
- **dlh** (*lm_max*) *[real,in]*
- **lm_max** *[integer,in,]*
- **l_max** *[integer,in]*
- **minc** *[integer,in]*

- **r** *[real,in]*
- **r_icb** *[real,in]*
- **lderiv** *[logical,in]*
- **lhor** *[logical,in]*
- **lcondic** *[logical,in]*
- **dlhw** (*lm_max*) *[complex,out]*
- **vhg** (*lm_max*) *[complex,out]*
- **vhc** (*lm_max*) *[complex,out]*
- **dlhz** (*lm_max*) *[complex,out]*
- **cvhg** (*lm_max*) *[complex,out]*
- **cvhc** (*lm_max*) *[complex,out]*

Called from *store_movie_frame_ic()*, *graphout_ic()*

10.10.3 legendre_spec_to_grid.f90

Quick access

Routines *lmas2pt()*, *legtf()*, *legtfg()*, *legtfgnomag()*

Needed modules

- *horizontal_data* (*dplm()*, *d_mc2m()*, *lstop()*, *lmodd()*, *osn2()*, *plm()*, *lstart()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *parallel_mod* (rank()): This module contains the blocking information
- *precision_mod*: This module controls the precision used in MagIC
- *logic* (*l_heat()*, *l_ht()*): Module containing the logicals that control the run
- *leg_helper_mod* (leg_helper_t())
- *blocking* (*lm2mc()*, *lm2()*, *nfs()*, *sizethetab()*): Module containing blocking information
- *constants* (*zero()*, *one()*, *half()*): module containing constants and parameters used in the code.
- *truncation* (*lm_max()*, *l_max()*, *nrp()*, *n_m_max()*): This module defines the grid points and the truncation

Variables

Subroutines and functions

subroutine legendre_spec_to_grid/**legtfg** (*nbc*, *lderiv*, *lviscbccalc*, *lpresscalc*, *nthetastart*, *vrc*, *vtc*, *vpc*, *dvrdrc*, *dvtdrc*, *dvpdrc*, *cvrc*, *dvrdtc*, *dvrdpc*, *dvtdpc*, *dvpdpc*, *brc*, *btc*, *bpc*, *cbrc*, *cbtc*, *cbpc*, *sc*, *drsc*, *dsdtc*, *dsdpc*, *pc*, *leg_helper*)

Legendre transform from (nR,l,m) to (nR,nTheta,m) [spectral to grid] where nTheta numbers the colatitudes and l is the degree of the spherical harmonic representation.

Transforms entropy, velocity and magnetic field components and terms involving spatial derivatives. The symmetry properties of the P_lm with respect to the equator are used. The equatorially anti-symmetric (EA) contribution is added to (subracted from) the equatorially symmetric (ES) contribution in northern (southern) hemisphere.

•nBc : (input) accounts for special conditions on radial boundaries

-nBc=2 [we are dealing with a no slip boundary, v_r and v_theta are] zero and v_phi=r sin(theta) omega, where omega is the rotation rate of the boundary (mantle of IC), only magn. field terms are calculated, v is set later.

-nBc=1 [a free slip bounday: v_r is zero, derivatives of v and B] are not needed, only components of v,B and entropy are calculated

-nBc=0 : normal case, interior grid point

•lDeriv=.true. : (input) calculate derivatives

•nThetaStart : (input) transformation is done for the range of points nThetaStart <= nTheta <= nThetaStart-1+sizeThetaB

•Plm : associated Legendre polynomials

•dPlm : sin(theta) d Plm / d theta

•osn2 : 1/sin(theta)^2

•vrc, ..., drSc: (output) components in (nTheta,m)-space

•dLhw,....,cbhC : (input) help arrays calculated in s_legPrep.f

Parameters

- **nbc** *[integer,in]*
- **lderiv** *[logical,in]*
- **lviscbccalc** *[logical,in]*
- **lpresscalc** *[logical,in]*
- **nthetastart** *[integer,in]*
- **vrc** (*nrp*,*nfs*) *[real,out]*
- **vtc** (*nrp*,*nfs*) *[real,out]*
- **vpc** (*nrp*,*nfs*) *[real,out]*
- **dvrdrc** (*nrp*,*nfs*) *[real,out]*
- **dvtdrc** (*nrp*,*nfs*) *[real,out]*
- **dvpdrc** (*nrp*,*nfs*) *[real,out]*
- **cvrc** (*nrp*,*nfs*) *[real,out]*
- **dvrdtc** (*nrp*,*nfs*) *[real,out]*
- **dvrdpc** (*nrp*,*nfs*) *[real,out]*
- **dvtdpc** (*nrp*,*nfs*) *[real,out]*
- **dvpdpc** (*nrp*,*nfs*) *[real,out]*

- **brc** (*nrp*,*nfs*) *[real,out]*
- **btc** (*nrp*,*nfs*) *[real,out]*
- **bpc** (*nrp*,*nfs*) *[real,out]*
- **cbrc** (*nrp*,*nfs*) *[real,out]*
- **cbtc** (*nrp*,*nfs*) *[real,out]*
- **cbpc** (*nrp*,*nfs*) *[real,out]*
- **sc** (*nrp*,*nfs*) *[real,out]*
- **drsc** (*nrp*,*nfs*) *[real,out]*
- **dsdtc** (*nrp*,*nfs*) *[real,out]*
- **dsdpc** (*nrp*,*nfs*) *[real,out]*
- **pc** (*nrp*,*nfs*) *[real,out]*
- **leg_helper** *[leg_helper_t]*

Called from *transform_to_grid_space()*

subroutine legendre_spec_to_grid/**legtfgnomag**(*nbc*, *lderiv*, *lviscbccalc*, *lpresscalc*, *nthetastart*, *vrc*, *vtc*, *vpc*, *dvrdrc*, *dvtdrc*, *dvpdrc*, *cvrc*, *dvrdtc*, *dvrdpc*, *dvtdpc*, *dvpdpc*, *sc*, *drsc*, *dsdtc*, *dsdpc*, *pc*, *leg_helper*)

Same as legTFG for non-magnetic cases

Parameters

- **nbc** *[integer,in]*
- **lderiv** *[logical,in]*
- **lviscbccalc** *[logical,in]*
- **lpresscalc** *[logical,in]*
- **nthetastart** *[integer,in]*
- **vrc** (*nrp*,*nfs*) *[real,out]*
- **vtc** (*nrp*,*nfs*) *[real,out]*
- **vpc** (*nrp*,*nfs*) *[real,out]*
- **dvrdrc** (*nrp*,*nfs*) *[real,out]*
- **dvtdrc** (*nrp*,*nfs*) *[real,out]*
- **dvpdrc** (*nrp*,*nfs*) *[real,out]*
- **cvrc** (*nrp*,*nfs*) *[real,out]*
- **dvrdtc** (*nrp*,*nfs*) *[real,out]*
- **dvrdpc** (*nrp*,*nfs*) *[real,out]*
- **dvtdpc** (*nrp*,*nfs*) *[real,out]*
- **dvpdpc** (*nrp*,*nfs*) *[real,out]*
- **sc** (*nrp*,*nfs*) *[real,out]*
- **drsc** (*nrp*,*nfs*) *[real,out]*

- **dsdtc** (*nrp*,*nfs*) *[real,out]*
- **dsdpc** (*nrp*,*nfs*) *[real,out]*
- **pc** (*nrp*,*nfs*) *[real,out]*
- **leg_helper** *[leg_helper_t]*

Called from `transform_to_grid_space()`

subroutine legendre_spec_to_grid/**legtf**(*dllhw*, *vhg*, *vhc*, *dllhz*, *cvhg*, *cvhc*, *l_max*, *minc*, *nthetastart*, *sizethetab*, *plm*, *dplm*, *lhor*, *lderiv*, *vrc*, *vtc*, *vpc*, *cvrc*, *cvtc*, *cvpc*)

'Legendre transform' from (nR,l,m) to (nR,nTheta,m) [spectral to grid] where nTheta numbers the colatitudes and l and m are degree and order of the spherical harmonic representation.

Calculates all three spherical components vrc,vtc,vpc of a field as well as its curl (cvrc,cvtc,cvpc) that is given a spherical harmonis poloidal toroidal decomposition. s_legPrep.f has to be called first and provides the input fields dLhW, ... The symmetry properties of the P_lm with respect to the equator are used. The equatorially anti-symmetric (EA) contribution is added to (subracted from) the equatorially symmetric (ES) contribution in northern (southern) hemisphere.

Output is given for all sizeThetaB colatitudes in a colatitude block that starts with colatitude nThetaStart. At output, each component in the northern hemisphere is followed by the component in the southern hemisphere. The Plms and dPlms=sin(theta) d Plm / d theta are only given for the colatitudes in the northern hemisphere.

•dLhw,....,cvhC : (input) arrays provided by s_legPrep.f

•l_max : (input) maximum spherical harmonic degree

•minc : (input) azimuthal symmetry

•nThetaStart : (input) transformation is done for the range of points nThetaStart <= nTheta <= nThetaStart-1+sizeThetaB

•sizeThetaB : (input) size theta block

•Plm : (input) associated Legendre polynomials

•dPlm : (input) sin(theta) d Plm / d theta

•lHor=.true. : (input) calculate horizontal componenst

•lDeriv=.true. : (input) calculate curl of field

•vrc, ...,cvpc : (output) components in (nTheta,m)-space

Parameters

- **dllhw** (*) *[complex,in]*
- **vhg** (*) *[complex,in]*
- **vhc** (*) *[complex,in]*
- **dllhz** (*) *[complex,in]*
- **cvhg** (*) *[complex,in]*
- **cvhc** (*) *[complex,in]*
- **l_max** *[integer,in]*
- **minc** *[integer,in]*

- **nthetastart** *[integer,in]*
- **sizethetab** *[integer,in]*
- **plm** (*lm_max*,*) *[real,in]*
- **dplm** (*lm_max*,*) *[real,in]*
- **lhor** *[logical,in]*
- **lderiv** *[logical,in]*
- **vrc** (*nrp*,*) *[real,out]*
- **vtc** (*nrp*,*) *[real,out]*
- **vpc** (*nrp*,*) *[real,out]*
- **cvrc** (*nrp*,*) *[real,out]*
- **cvtc** (*nrp*,*) *[real,out]*
- **cvpc** (*nrp*,*) *[real,out]*

Called from *fields_average()*, *store_movie_frame_ic()*, *graphout_ic()*

subroutine legendre_spec_to_grid/**lmas2pt**(*alm*, *aij*, *nthetastart*, *nthetablocksize*)

Spherical harmonic transform from alm(l) to aij(theta) Done within the range [nThetaStart,n_thetaStart+nThetaBlockSize-1] only considering axisymmetric contributions. alm contains only m=0 coefficients

Parameters

- **alm** (*) *[real,in]* :: field in (l,m)-space
- **aij** (*) *[real,out]* :: field in (theta,phi)-space
- **nthetastart** *[integer,in]* :: first theta to be treated
- **nthetablocksize** *[integer,in]*

Called from *outperppar()*, *outmisc()*, *outpar()*

10.10.4 legendre_grid_to_spec.f90

Quick access

Routines *legtfas2()*, *legtf1()*, *legtf2()*, *legtf3()*, *legtfas()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *horizontal_data* (*wplm()*, *lmoddp()*, *lstopp()*, *lstartp()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *blocking* (*nfs()*, *sizethetab()*): Module containing blocking information
- *truncation* (*n_theta_max()*, *nrp()*, *n_m_max()*, *lmp_max()*): This module defines the grid points and the truncation

Variables

Subroutines and functions

subroutine `legendre_grid_to_spec/`**`legtf1`**(*nthetastart*, *f1lm*, *f1tm*)

Legendre transform (n_r,n_theta,m) to (n_r,l,m) [grid to spectral] for 2 arrays f1TM (input) to f1LM (output) One call to this routine does part of the transform by summation over theta points in one theta block: nThetaStart,..,nThetaStart+n_theta_block-1

Parameters

- **nthetastart** *[integer,in]* :: First no of theta on block
- **f1lm** (*lmp_max*) *[complex,inout]*
- **f1tm** (*nrp*,*nfs*) *[real,in]*

Called from *transform_to_lm_space()*, *initv()*, *inits()*

subroutine `legendre_grid_to_spec/`**`legtf2`**(*nthetastart*, *f1lm*, *f2lm*, *f1tm*, *f2tm*)

Legendre transform (n_r,n_theta,m) to (n_r,l,m) [grid to spectral] for 2 arrays f1TM,f2TM (input) to f1LM,f2LM (output) One call to this routine does part of the transform by summation over theta points in on theta block: nThetaStart,..,nThetaStart+n_theta_block-1

Parameters

- **nthetastart** *[integer,in]* :: First no of theta on block
- **f1lm** (*lmp_max*) *[complex,inout]*
- **f2lm** (*lmp_max*) *[complex,inout]*
- **f1tm** (*nrp*,*nfs*) *[real,in]*
- **f2tm** (*nrp*,*nfs*) *[real,in]*

Called from *transform_to_lm_space()*, *get_dtblm()*, *get_br_v_bcs()*

subroutine `legendre_grid_to_spec/`**`legtf3`**(*nthetastart*, *f1lm*, *f2lm*, *f3lm*, *f1tm*, *f2tm*, *f3tm*)

Legendre transform (n_r,n_theta,m) to (n_r,l,m) [grid to spectral] for 2 arrays ancl1/2/3 (input) to flm1/2/3 (output) One call to this routine does part of the transform by summation over theta points in on theta block: nThetaStart,..,nThetaStart+n_theta_block-1

Parameters

- **nthetastart** *[integer,in]* :: First no of theta on block
- **f1lm** (*lmp_max*) *[complex,inout]*
- **f2lm** (*lmp_max*) *[complex,inout]*
- **f3lm** (*lmp_max*) *[complex,inout]*
- **f1tm** (*nrp*,*nfs*) *[real,in]*
- **f2tm** (*nrp*,*nfs*) *[real,in]*
- **f3tm** (*nrp*,*nfs*) *[real,in]*

Called from *transform_to_lm_space()*, *get_dtblm()*

subroutine legendre_grid_to_spec/**legtfas** (*flm1*, *ft1*, *lmmax*, *nthetastart*, *sizethetab*)

Legendre transform (n_r,n_theta,m) to (n_r,l,m) [grid to spectral] for 2 arrays ancl1/2 (input) to flm1/2 (output) One call to this routine does part of the transform by summation over theta points in on theta block: n_theta_min,..,n_theta_min+n_theta_block-1

Parameters

- **flm1** (*) *[real,out]*
- **ft1** (*) *[real,in]*
- **lmmax** *[integer,in]* :: Number of modes to be processed
- **nthetastart** *[integer,in]* :: First no of theta on block
- **sizethetab** *[integer,in]* :: Size of theta block

Called from *get_fluxes()*, *get_nlblayers()*, *outto()*

subroutine legendre_grid_to_spec/**legtfas2** (*flm1*, *flm2*, *ft1*, *ft2*, *lmmax*, *nthetastart*, *sizethetab*)

Legendre transform (n_r,n_theta,m) to (n_r,l,m) [grid to spectral] for 2 arrays ancl1/2 (input) to flm1/2 (output) One call to this routine does part of the transform by summation over theta points in on theta block: n_theta_min,..,n_theta_min+n_theta_block-1

Parameters

- **flm1** (*) *[real,out]*
- **flm2** (*) *[real,out]*
- **ft1** (*) *[real,in]*
- **ft2** (*) *[real,in]*
- **lmmax** *[integer,in]* :: Number of modes to be processed
- **nthetastart** *[integer,in]* :: First no of theta on block
- **sizethetab** *[integer,in]* :: Size of theta block

Called from *get_fluxes()*, *get_helicity()*, *get_nlblayers()*, *getto()*, *get_perppar()*, *outto()*

10.11 Fourier transforms

10.11.1 fft.f90

Description

This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5

Quick access

Variables *nd*, *ni*, *i_fft_init*, *d_fft_init*

Routines *wpass3jw()*, *wpass4jw()*, *wpass5jw()*, *wpass2jw()*, *fftjw()*, *fft99ajw()*, *fft_thetab()*, *fft_to_real()*, *init_fft()*, *fft99bjw()*

Needed modules

- *parallel_mod*: This module contains the blocking information
- *useful* (*factorise()*): library with several useful subroutines
- *precision_mod*: This module controls the precision used in MagIC
- *blocking*: Module containing blocking information
- *constants* (*cos36()*, *sin36()*, *cos72()*, *two()*, *one()*, *sin72()*, *sin60()*, *half()*, *pi()*): module containing constants and parameters used in the code.
- *truncation*: This module defines the grid points and the truncation

Variables

- fft/**ni** *[integer,private/parameter=100]*
- fft/**nd** *[integer,private]*
- fft/**d_fft_init** (:) *[real,private/allocatable]*
- fft/**i_fft_init** (100) *[integer,private]*

Subroutines and functions

subroutine fft/**init_fft**(*n*)

Purpose of this subroutine is to calculate and store several values that will be needed for a fast fft transform. The actual transform is performed by the subroutine fftJW.

Parameters **n** *[integer,in]* :: Dimension of problem, number of grid points

Called from *horizontal()*

Call to *factorise()*

subroutine fft/**fft_to_real**(*f*, *ld_f*, *nrep*)

Parameters

- **f** (ld_f,nrep) *[real,inout]*
- **ld_f** *[integer,in,]*
- **nrep** *[integer,in,]*

Called from *getdvptr()*, *getpvptr()*

Call to *fftjw()*

subroutine fft/**fft_thetab**(*f*, *dir*)

Parameters

- **f** (*nrp*,*nfs*) *[real,inout]*
- **dir** *[integer,in]* :: back or forth transform

Called from *transform_to_lm_space()*, *fields_average()*, *initv()*, *inits()*, *get_btor()*, *transform_to_grid_space()*, *get_b_surface()*, *store_movie_frame_ic()*, *get_bpol()*, *get_dtblm()*, *lm2pt()*, *get_br_v_bcs()*, *graphout_ic()*

Call to *fftjw()*

subroutine `fft/` **`fftjw`** (*a*, *ld_a*, *n*, *isign*, *nsize*, *wrk*, *wd1*, *wd2*, *i_fft_init*, *d_fft_init*)

The routines has been adopted by Gary Glatzmaier and has subsequently been modified by Uli Christensen and Johannes Wicht

It performs a number of simultaneous real/half-complex periodic fast fourier transforms or corresponding inverse transforms, using ordinary spatial order of gridpoint values. given a set of real data vectors, the package returns a set of "half-complex" fourier coefficient vectors, or vice versa. the length of the transforms must be an even number that has no other factors except possibly powers of 2, 3, and 5. this version of fft991 is optimized for use on the cray-1.

on input a(ld_a,*)

an array of length (ld_a,nsize) containing the input data or coefficient vectors. This array is overwritten by the results.

n the length of each transform (see definition of transforms, below).

nsize the number of transforms to be done simultaneously.

isign

= +1 for a transform from fourier coefficients to gridpoint values.

= -1 for a transform from gridpoint values to fourier coefficients.

on output a if isign = +1, and n_theta_max coefficient vectors are supplied each containing the sequence

a(0),b(0),a(1),b(1),...,a(n/2),b(n/2) (n+2 values)

then the result consists of n_theta_max data vectors each containing the corresponding n+2 gridpoint values

for fft991, x(0), x(1), x(2),...,x(n-1),0,0. (n+2) real values with x(n)=x(n+1)=0

when isign = +1, the transform is defined by x(j)=sum(k=0,...,n-1)(c(k)*exp(2*i*j*k*pi/n)) where c(k)=a(k)+i*b(k) and c(n-k)=a(k)-i*b(k) and i=sqrt (-1) for k=0,...,n/2 i.e., (n/2+1) complex values with c(0) = c(n) = a(0) and c(n/2)=a(n/2)=0

if isign = -1, and n_theta_max data vectors are supplied each containing a sequence of gridpoint values x(j) as defined above, then the result consists of n_theta_max vectors each containing the corresponding fourier cofficients a(k), b(k), 0 <= k .le n/2.

when isign = -1, the inverse transform is defined by c(k)=(1/n)*sum(j=0,...,n-1)(x(j)*exp(-2*i*j*k*pi/n)) where c(k)=a(k)+i*b(k) and i=sqrt(-1) for k=0,...,n/2

a call with isign=+1 followed by a call with isign=-1 (or vice versa) returns the original data.

note the fact that the gridpoint values x(j) are real implies that b(0)=b(n/2)=0. for a call with isign=+1, it is not actually necessary to supply these zeros. note starting from grid with x(n)=x(n+1)=0 then transforming to spectral (sign=-1) then c(n/2)=a(n/2) is not necessarily 0 unless there is no aliasing.

Parameters

- **a** (ld_a,*) *[real,inout]* :: fields to be transformed
- **ld_a** *[integer,in,]* :: leading dimension of a
- **n** *[integer,in]* :: dimension of problem
- **isign** *[integer,in]* :: back/forth transtorm for isign=1/-1
- **nsize** *[integer,in]* :: number of fields for
- **wrk** (wd1,wd2) *[real,inout]* :: work array
- **wd1** *[integer,in,]*
- **wd2** *[integer,in,]*
- **i_fft_init** (*) *[integer,in]* :: factorization information from init_fft
- **d_fft_init** (*) *[real,in]* :: trigonometric functions from init_fft

Called from *fft_thetab()*, *fft_to_real()*

Call to *fft99ajw()*, *wpass2jw()*, *wpass3jw()*, *wpass4jw()*, *wpass5jw()*, *fft99bjw()*

subroutine fft/**fft99ajw**(*a*, *work*, *trigs*, *nrp*, *nsize*)

Parameters

- **a** (*) *[real,inout]*
- **work** (*) *[real,inout]*
- **trigs** (*) *[real,in]*
- **nrp** *[integer,in]*
- **nsize** *[integer,in]*

Called from *fftjw()*

subroutine fft/**fft99bjw**(*work*, *a*, *trigsf*, *nrp*, *nsize*)

postprocessing step (isign=-1) (gridpoint to spectral transform)

Parameters

- **work** (*) *[real,inout]*
- **a** (*) *[real,inout]*
- **trigsf** (*) *[real,in]*
- **nrp** *[integer,in]*
- **nsize** *[integer,in]*

Called from *fftjw()*

subroutine fft/**wpass2jw**(*a*, *b*, *c*, *d*, *trigs*, *nrp*, *nsize*)

reduction for factor 2

if(la /= 1) stop 'call to wpass2 with la /= 1'

Parameters

- **a** (*) *[real,in]*

- **b** (*) *[real,in]*
- **c** (*) *[real,out]*
- **d** (*) *[real,out]*
- **trigs** (*) *[real,in]*
- **nrp** *[integer,in]*
- **nsize** *[integer,in]*

Called from *fftjw()*

subroutine `fft/`**`wpass3jw`**(*a*, *b*, *c*, *d*, *trigs*, *nrp*, *la*, *nsize*)

called in fftJW

Parameters

- **a** (*) *[real,in]*
- **b** (*) *[real,in]*
- **c** (*) *[real,out]*
- **d** (*) *[real,out]*
- **trigs** (*) *[real,in]*
- **nrp** *[integer,in]*
- **la** *[integer,in]*
- **nsize** *[integer,in]*

Called from *fftjw()*

subroutine `fft/`**`wpass4jw`**(*a*, *b*, *c*, *d*, *trigs*, *nrp*, *la*, *nsize*)

called in fftJW reduction for factor 4

Parameters

- **a** (*) *[real,in]*
- **b** (*) *[real,in]*
- **c** (*) *[real,out]*
- **d** (*) *[real,out]*
- **trigs** (*) *[real,in]*
- **nrp** *[integer,in]*
- **la** *[integer,in]*
- **nsize** *[integer,in]*

Called from *fftjw()*

subroutine `fft/`**`wpass5jw`**(*a*, *b*, *c*, *d*, *trigs*, *nrp*, *la*, *nsize*)

called in fftJW reduction for factor 5

Parameters

- **a** (*) *[real,in]*
- **b** (*) *[real,in]*
- **c** (*) *[real,out]*
- **d** (*) *[real,out]*
- **trigs** (*) *[real,in]*
- **nrp** *[integer,in]*
- **la** *[integer,in]*
- **nsize** *[integer,in]*

Called from *fftjw()*

10.12 Linear algebra

10.12.1 algebra.f90

Quick access

Variables *zero_tolerance*

Routines *sgesl()*, *cgeslml()*, *cgesl()*, *sgefa()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- omp_lib
- *constants* (*one()*): module containing constants and parameters used in the code.

Variables

- algebra/**zero_tolerance** *[real,private/parameter=1.0e-15_cp]*

Subroutines and functions

subroutine algebra/**cgesl**(*a*, *ia*, *n*, *ip*, *bc1*)

This routine does the backward substitution into a lu-decomposed real matrix a (to solve a * x = bc1) were bc1 is the right hand side vector. On return x is stored in bc1.

Parameters

- **a** (ia,*) *[real,in]* :: real n X n matrix
- **ia** *[integer,in,]* :: first dim of a
- **n** *[integer,in]* :: dimension of problem
- **ip** (*) *[integer,in]* :: pivot pointer of legth n

- **bc1** (*) *[complex,inout]* :: on input RHS of problem

Called from *j_cond()*, *updatez()*

subroutine algebra/**cgeslml**(*a*, *ia*, *n*, *ip*, *bc*, *ldbc*, *nrhss*)

This routine does the backward substitution into a lu-decomposed real matrix a (to solve a * x = bc) simultaneously for nRHSs complex vectors bc. On return the results are stored in the bc.

Parameters

- **a** (ia,n) *[real,in]* :: real n X n matrix
- **ia** *[integer,in,]* :: leading dimension of a
- **n** *[integer,in,]* :: dimension of problem
- **ip** (n) *[integer,in]* :: pivot pointer of length n
- **bc** (ldbc,nrhss) *[complex,inout]* :: on input RHS of problem
- **ldbc** *[integer,in,]* :: leading dimension of bc
- **nrhss** *[integer,in,]* :: number of right-hand sides

Called from *updatez()*, *updateb()*, *updates()*, *updatewp()*, *updates_ala()*

subroutine algebra/**sgesl**(*a*, *ia*, *n*, *ip*, *b*)

like the linpack routine backward substitution of vector b into lu-decomposed matrix a to solve a * x = b for a single real vector b

sub sgefa must be called once first to initialize a and ip

a: (input) nxn real matrix n: (input) size of a and b ip: (input) pivot pointer array of length n b: (in/output) rhs-vector on input, solution on output

Parameters

- **a** (ia,*) *[real,in]*
- **ia** *[integer,in,]* :: first dim of a
- **n** *[integer,in]* :: dim of problem
- **ip** (*) *[integer,in]* :: pivot information
- **b** (*) *[real,inout]*

Called from *inits()*, *updates()*, *s_cond()*, *getbackground()*, *updatewp()*, *updates_ala()*

subroutine algebra/**sgefa**(*a*, *ia*, *n*, *ip*, *info*)

like the linpack routine

lu decomposes the real matrix a(n,n) via gaussian elimination

a: (in/output) real nxn matrix on input, lu-decomposed matrix on output ia: (input) first dimension of a (must be >= n) n: (input) 2nd dimension and rank of a ip: (output) pivot pointer array info: (output) error message when /= 0

Parameters

- **a** (ia,*) *[real,inout]*

- **ia** *[integer,in,]*
- **n** *[integer,in]*
- **ip** (*) *[integer,out]* :: pivoting information
- **info** *[integer,out]*

Called from *j_cond()*, *get_z10mat()*, *inits()*, *s_cond()*, *get_bmat()*, *getbackground()*, *get_p0mat()*, *get_s0mat()*, *get_smat()*, *get_zmat()*, *get_wpmat()*

10.13 Radial derivatives and integration

10.13.1 radial_derivatives.f90

Description

Radial derivatives functions

Quick access

Variables *get_dcheb*, *get_dr*

Routines *get_dddcheb()*, *get_dddr()*, *get_ddcheb()*, *get_ddr()*, *get_dcheb_complex()*, *get_drns()*, *get_dr_complex()*, *get_dcheb_real_1d()*, *get_dr_real_1d()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *cosine_transform_odd*
- *constants* (*zero()*, *three()*, *one()*): module containing constants and parameters used in the code.

Variables

- radial_der/**get_dcheb** *[public]*
- radial_der/**get_dr** *[public]*

Subroutines and functions

subroutine radial_der/**get_dr_complex** (*f*, *df*, *n_f_max*, *n_f_start*, *n_f_stop*, *n_r_max*, *n_cheb_max*, *work1*, *work2*, *chebt*, *drx*)

Returns first radial derivative df of the input function f. Array f(n_f_max,*) may contain several functions numbered by the first index. The subroutine calculates the derivaties of the functions f(n_f_start,*) to f(n_f_stop) by transforming to a Chebychev representation using n_r_max radial grid points .

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,in]*
- **df** (n_f_max,*n_r_max*) *[complex,out]* :: first derivative of f
- **n_f_max** *[integer,in,]* :: first dim of f
- **n_f_start** *[integer,in]* :: first function to be treated
- **n_f_stop** *[integer,in]* :: last function to be treated
- **n_r_max** *[integer,in,]* :: number of radial grid points
- **n_cheb_max** *[integer,in]* :: max number of cheb modes
- **work1** (n_f_max,*n_r_max*) *[complex,out]* :: work array needed for costf
- **work2** (n_f_max,*n_r_max*) *[complex,out]* :: work array for f transfer
- **chebt** *[costf_odd_t,in]*
- **drx** (*n_r_max*) *[real,in]* :: first derivatives of x(r)

subroutine `radial_der/`**`get_dr_real_1d`**(*f*, *df*, *n_r_max*, *n_cheb_max*, *work1*, *work2*, *chebt*, *drx*)

Parameters

- **f** (*n_r_max*) *[real,in]*
- **df** (*n_r_max*) *[real,out]* :: first derivative of f
- **n_r_max** *[integer,in,]* :: number of radial grid points
- **n_cheb_max** *[integer,in]* :: max number of cheb modes
- **work1** (*n_r_max*) *[real,out]* :: work array needed for costf
- **work2** (*n_r_max*) *[real,out]* :: work array for f transfer
- **chebt** *[costf_odd_t,in]*
- **drx** (*n_r_max*) *[real,in]* :: first derivatives of x(r)

subroutine `radial_der/`**`get_drns`**(*f*, *df*, *n_f_max*, *n_f_start*, *n_f_stop*, *n_r_max*, *n_cheb_max*, *work1*, *chebt*, *drx*)

Returns first radial derivative df of the input function f. Array f(n_f_max,*) may contain several functions numbered by the first index. The subroutine calculates the derivatives of the functions f(n_f_start,*) to f(n_f_stop,*) by transforming to a Chebychev representation using n_r_max radial grid points . Note: when using this function the input field f is slightly changed by the back and forth transform. Use s_get_dr.f to avoid this.

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,inout]*
- **df** (n_f_max,*n_r_max*) *[complex,out]* :: first derivative of f
- **n_f_max** *[integer,in,]* :: first dim of f
- **n_f_start** *[integer,in]* :: first function to be treated
- **n_f_stop** *[integer,in]* :: last function to be treated
- **n_r_max** *[integer,in,]* :: number of radial grid points
- **n_cheb_max** *[integer,in]* :: max number of cheb modes
- **work1** (n_f_max,*n_r_max*) *[complex,out]* :: work array needed for costf

- **chebt** *[costf_odd_t,in]*
- **drx** (*) *[real,in]* :: first derivatives of x(r)

Called from *fields_average()*, *updateb()*, *updates()*, *dtvrms()*, *dtbrms()*, *get_dtblmfinish()*, *updates_ala()*, *write_dtb_frame()*

subroutine radial_der/**get_ddr** (*f*, *df*, *ddf*, *n_f_max*, *n_f_start*, *n_f_stop*, *n_r_max*, *n_cheb_max*, *work1*, *work2*, *chebt*, *drx*, *ddrx*)

Returns first radial derivative df and second radial derivative ddf of the input function f. Array f(n_f_max,*) may contain several functions numbered by the first index. The subroutine calculates the derivatives of the functions f(n_f_start,*) to f(n_f_stop) by transforming to a Chebychev representation using n_r_max radial grid points.

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,in]*
- **df** (n_f_max,*n_r_max*) *[complex,out]* :: first derivative of f
- **ddf** (n_f_max,*n_r_max*) *[complex,out]* :: second derivative of f
- **n_f_max** *[integer,in,]* :: first dim of f
- **n_f_start** *[integer,in]* :: first function to be treated
- **n_f_stop** *[integer,in]* :: last function to be treated
- **n_r_max** *[integer,in,]* :: number of radial grid points
- **n_cheb_max** *[integer,in]* :: number of cheb modes
- **work1** (n_f_max,*n_r_max*) *[complex,out]* :: work array needed for costf
- **work2** (n_f_max,*n_r_max*) *[complex,out]* :: work array for f transfer
- **chebt** *[costf_odd_t,in]*
- **drx** (*n_r_max*) *[real,in]* :: first derivatives of x(r)
- **ddrx** (*n_r_max*) *[real,in]* :: second derivatives of x(r)

Called from *updatez()*, *updateb()*, *updates()*, *getstartfields()*, *updates_ala()*

Call to *get_ddcheb()*

subroutine radial_der/**get_dddr** (*f*, *df*, *ddf*, *dddf*, *n_f_max*, *n_f_start*, *n_f_stop*, *n_r_max*, *n_cheb_max*, *work1*, *work2*, *chebt*, *drx*, *ddrx*, *dddrx*)

Returns first radial derivative df, the second radial deriv. ddf, and the third radial derivative dddf of the input function f. Array f(n_f_max,*) may contain several functions numbered by the first index. The subroutine calculates the derivaties of the functions f(n_f_start,*) to f(n_f_stop) by transforming to a Chebychev representation using n_r_max radial grid points.

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,in]*
- **df** (n_f_max,*n_r_max*) *[complex,out]* :: first derivative of f
- **ddf** (n_f_max,*n_r_max*) *[complex,out]* :: second derivative of f
- **dddf** (n_f_max,*n_r_max*) *[complex,out]* :: third derivative of f

- **n_f_max** *[integer,in,]* :: first dim of f
- **n_f_start** *[integer,in]* :: first function to be treated
- **n_f_stop** *[integer,in]* :: last function to be treated
- **n_r_max** *[integer,in,]* :: number of radial grid points
- **n_cheb_max** *[integer,in]* :: number of cheb_modes
- **work1** (n_f_max,*n_r_max*) *[complex,out]* :: work array needed for costf
- **work2** (n_f_max,*n_r_max*) *[complex,out]* :: work array needed for costf
- **chebt** *[costf_odd_t,in]*
- **drx** (*n_r_max*) *[real,in]* :: first derivatives of x(r)
- **ddrx** (*n_r_max*) *[real,in]* :: second derivatives of x(r)
- **dddrx** (*n_r_max*) *[real,in]* :: third derivatives of x(r)

Called from *updatewp()*

Call to *get_dddcheb()*

subroutine radial_der/**get_dcheb_complex**(*f*, *df*, *n_f_max*, *n_f_start*, *n_f_stop*, *n_r_max*, *n_cheb_max*, *d_fac*)

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,in]*
- **df** (n_f_max,*n_r_max*) *[complex,out]*
- **n_f_max** *[integer,in,]* :: Max no of functions
- **n_f_start** *[integer,in]* :: No of function to start with
- **n_f_stop** *[integer,in]* :: No of function to stop with
- **n_r_max** *[integer,in,]* :: second dimension of f,df,ddf
- **n_cheb_max** *[integer,in]* :: Number of cheb modes
- **d_fac** *[real,in]* :: factor for interval mapping

subroutine radial_der/**get_dcheb_real_1d**(*f*, *df*, *n_r_max*, *n_cheb_max*, *d_fac*)

Parameters

- **f** (*n_r_max*) *[real,in]*
- **df** (*n_r_max*) *[real,out]*
- **n_r_max** *[integer,in,]* :: Dimension of f,df,ddf
- **n_cheb_max** *[integer,in]* :: Number of cheb modes
- **d_fac** *[real,in]* :: factor for interval mapping

subroutine radial_der/**get_ddcheb**(*f*, *df*, *ddf*, *n_f_max*, *n_f_start*, *n_f_stop*, *n_r_max*, *n_cheb_max*, *d_fac*)

Returns chebychev coefficents of first derivative df and second derivative ddf for a function whose cheb-coeff. are given as columns in array f(n_c_tot,n_r_max).

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,in]*

- **df** (n_f_max,*n_r_max*) *[complex,out]*
- **ddf** (n_f_max,*n_r_max*) *[complex,out]*
- **n_f_max** *[integer,in,]* :: First dimension of f,df,ddf
- **n_f_start** *[integer,in]* :: No of column to start with
- **n_f_stop** *[integer,in]* :: No of column to stop with
- **n_r_max** *[integer,in,]* :: second dimension of f,df,ddf
- **n_cheb_max** *[integer,in]* :: Number of cheb modes
- **d_fac** *[real,in]* :: factor for interval mapping

Called from *get_ddr()*

subroutine radial_der/**get_dddcheb**(*f*, *df*, *ddf*, *dddf*, *n_f_max*, *n_f_start*, *n_f_stop*, *n_r_max*, *n_cheb_max*, *d_fac*)

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,in]*
- **df** (n_f_max,*n_r_max*) *[complex,out]*
- **ddf** (n_f_max,*n_r_max*) *[complex,out]*
- **dddf** (n_f_max,*n_r_max*) *[complex,out]*
- **n_f_max** *[integer,in,]* :: First dimension of f,df,ddf
- **n_f_start** *[integer,in]* :: No of column to start with
- **n_f_stop** *[integer,in]* :: No of column to stop with
- **n_r_max** *[integer,in,]* :: second dimension of f,df,ddf
- **n_cheb_max** *[integer,in]* :: Number of cheb modes
- **d_fac** *[real,in]* :: factor for interval mapping

Called from *get_dddr()*

10.13.2 radial_derivatives_even.f90

Quick access

Routines *get_ddcheb_even()*, *get_ddrns_even()*, *get_ddr_even()*, *get_dcheb_even()*, *get_drns_even()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *cosine_transform_odd*
- *cosine_transform_even*
- *constants* (*zero()*): module containing constants and parameters used in the code.

Variables

Subroutines and functions

subroutine radial_der_even/**get_ddr_even**(*f*, *df*, *ddf*, *n_f_max*, *n_f_start*, *n_f_stop*, *n_r_max*, *n_cheb_max*, *dr_fac*, *work1*, *work2*, *chebt_odd*, *chebt_even*)

Returns first rarial derivative df and second radial derivative ddf of the input function f. Array f(n_f_max,*) may contain several functions numbered by the first index. The subroutine calculates the derivaties of the functions f(n_f_start,*) to f(n_f_stop) by transforming to a Chebychev representation using n_r_max radial grid points. The cheb transforms have to be initialized by calling init_costf1 and init_costf2.

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,in]*
- **df** (n_f_max,*n_r_max*) *[complex,out]* :: first derivative of f
- **ddf** (n_f_max,*n_r_max*) *[complex,out]* :: second derivative of f
- **n_f_max** *[integer,in,]* :: first dim of f
- **n_f_start** *[integer,in]* :: first function to be treated
- **n_f_stop** *[integer,in]* :: last function to be treated
- **n_r_max** *[integer,in,]* :: number of radial grid points
- **n_cheb_max** *[integer,in]* :: number of cheb modes
- **dr_fac** *[real,in]* :: mapping factor
- **work1** (n_f_max,*n_r_max*) *[complex,out]* :: work array needed for costf
- **work2** (n_f_max,*n_r_max*) *[complex,out]* :: work array needed for costf
- **chebt_odd** *[costf_odd_t,in]*
- **chebt_even** *[costf_even_t,in]*

Called from *updateb()*, *getstartfields()*

Call to *get_ddcheb_even()*

subroutine radial_der_even/**get_drns_even**(*f*, *df*, *n_f_max*, *n_f_start*, *n_f_stop*, *n_r_max*, *n_cheb_max*, *dr_fac*, *work1*, *chebt_odd*, *chebt_even*)

Returns first rarial derivative df and second radial derivative ddf of the input function f. Array f(n_f_max,*) may contain several functions numbered by the first index. The subroutine calculates the derivaties of the functions f(n_f_start,*) to f(n_f_stop) by transforming to a Chebychev representation using n_r_max radial grid points. The cheb transforms have to be initialized by calling init_costf1 and init_costf2.

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,inout]*
- **df** (n_f_max,*n_r_max*) *[complex,out]* :: first derivative of f
- **n_f_max** *[integer,in,]* :: first dim of f
- **n_f_start** *[integer,in]* :: first function to be treated

- **n_f_stop** *[integer,in]* :: last function to be treated
- **n_r_max** *[integer,in,]* :: number of radial grid points
- **n_cheb_max** *[integer,in]* :: number of cheb modes
- **dr_fac** *[real,in]* :: mapping factor
- **work1** (n_f_max,*n_r_max*) *[complex,out]* :: work array needed for costf
- **chebt_odd** *[costf_odd_t,in]*
- **chebt_even** *[costf_even_t,in]*

Called from *fields_average()*, *write_dtb_frame()*

Call to *get_dcheb_even()*

subroutine radial_der_even/**get_ddrns_even**(*f*, *df*, *ddf*, *n_f_max*, *n_f_start*, *n_f_stop*, *n_r_max*, *n_cheb_max*, *dr_fac*, *work1*, *chebt_odd*, *chebt_even*)

Returns first rarial derivative df and second radial derivative ddf of the input function f. Array f(n_f_max,*) may contain several functions numbered by the first index. The subroutine calculates the derivaties of the functions f(n_f_start,*) to f(n_f_stop) by transforming to a Chebychev representation using n_r_max radial grid points. The cheb transforms have to be initialized by calling init_costf1 and init_costf2.

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,inout]*
- **df** (n_f_max,*n_r_max*) *[complex,out]* :: first derivative of f
- **ddf** (n_f_max,*n_r_max*) *[complex,out]* :: second derivative of f
- **n_f_max** *[integer,in,]* :: first dim of f
- **n_f_start** *[integer,in]* :: first function to be treated
- **n_f_stop** *[integer,in]* :: last function to be treated
- **n_r_max** *[integer,in,]* :: number of radial grid points
- **n_cheb_max** *[integer,in]* :: number of cheb modes
- **dr_fac** *[real,in]* :: mapping factor
- **work1** (n_f_max,*n_r_max*) *[complex,out]* :: work array needed for costf
- **chebt_odd** *[costf_odd_t,in]*
- **chebt_even** *[costf_even_t,in]*

Called from *fields_average()*

Call to *get_ddcheb_even()*

subroutine radial_der_even/**get_dcheb_even**(*f*, *df*, *n_f_max*, *n_f_start*, *n_f_stop*, *n_r_max*, *n_cheb_max*, *d_fac*)

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,in]*
- **df** (n_f_max,*n_r_max*) *[complex,out]*
- **n_f_max** *[integer,in,]* :: First dimension of f,df

- **n_f_start** *[integer,in]* :: No of function to start with
- **n_f_stop** *[integer,in]* :: No of function to stop with
- **n_r_max** *[integer,in,]* :: second dimension of f,df
- **n_cheb_max** *[integer,in]* :: Number of cheb modes
- **d_fac** *[real,in]* :: factor for interval mapping

Called from *get_drns_even()*

subroutine radial_der_even/**get_ddcheb_even** (*f*, *df*, *ddf*, *n_f_max*, *n_f_start*, *n_f_stop*, *n_r_max*, *n_cheb_max*, *d_fac*)

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,in]*
- **df** (n_f_max,*n_r_max*) *[complex,out]*
- **ddf** (n_f_max,*n_r_max*) *[complex,out]*
- **n_f_max** *[integer,in,]* :: First dimension of f,df,ddf
- **n_f_start** *[integer,in]* :: No of function to start with
- **n_f_stop** *[integer,in]* :: No of function to stop with
- **n_r_max** *[integer,in,]* :: second dimension of f,df,ddf
- **n_cheb_max** *[integer,in]* :: Number of cheb modes
- **d_fac** *[real,in]* :: factor for interval mapping

Called from *get_ddr_even()*, *get_ddrns_even()*

10.13.3 integration.f90

Description

Radial integration functions

Quick access

Routines *rint_r()*, *rint()*, *rintic()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *cosine_transform_odd*
- *constants* (*one()*, *two()*, *half()*): module containing constants and parameters used in the code.

Variables

Subroutines and functions

function `integration/`**`rint`** (*f*, *nrmax*, *drfac*, *chebt*)

This function performs the radial integral over a function f that is given on the appropriate nRmax radial Chebychev grid points.

Note: drFac maps radius to cheb space [-1,1] drFac=two/(rMax-rMin)

Parameters

- **f** (nrmax) *[real,in]*
- **nrmax** *[integer,in,]*
- **drfac** *[real,in]*
- **chebt** *[costf_odd_t,in]*

Return **rint** *[real]*

Called from *outperppar()*, *outmisc()*, *dtbrms()*, *spectrum_temp_average()*, *spectrum_temp()*

function `integration/`**`rintic`** (*f*, *nrmax*, *drfac*, *chebt*)

This function performs the radial integral over a function f that is given on the appropriate nRmax radial Chebychev grid points.

Parameters

- **f** (nrmax) *[real,inout]*
- **nrmax** *[integer,in,]*
- **drfac** *[real,in]*
- **chebt** *[costf_odd_t,in]*

Return **rintic** *[real]*

Called from *spectrum()*, *get_power()*, *get_e_mag()*

function `integration/`**`rint_r`** (*f*, *n_r_max*, *n_cheb_max*, *dr_fac*, *chebt*)

Same as function rInt but for a radial dependent mapping function dr_fac2.

Parameters

- **f** (*n_r_max*) *[real,in]*
- **n_r_max** *[integer,in,]*
- **n_cheb_max** *[integer,in]*
- **dr_fac** (*n_r_max*) *[real,in]*
- **chebt** *[costf_odd_t,in]*

Return **rint_r** *[real]*

Called from *dtvrms()*, *output()*, *spectrum_average()*, *spectrum()*, *precalc()*, *get_angular_moment()*, *getdlm()*, *get_power()*, *get_u_square()*, *get_e_kin()*, *get_poltorrms()*, *get_e_mag()*, *outto()*

10.14 Blocking and LM mapping

10.14.1 `blocking.f90`

Description

Module containing blocking information

Quick access

Variables *get_theta_blocking*, *nfs*, *nlmbs*, *nthetabs*, *sizelmb*, *sizerb*, *sizethetab*, *nbsave*, *sizethetabi*, *cacheblock_size_in_b*, *nbdown*, *lmstartb*, *lmstopb*, *l2lmas*, *lm2l*, *lm2lma*, *lm2lmp*, *lm2lms*, *lm2m*, *lm2mc*, *lmp2l*, *lmp2lm*, *lmp2lmpa*, *lmp2lmps*, *nlmbs2*, *lm2*, *lmp2*, *sizelmb2*, *lm22l*, *lm22lm*, *lm22m*, *lo_map*, *lo_sub_map*, *sn_sub_map*, *st_map*, *st_sub_map*

Routines *initialize_blocking()*, *get_lorder_lm_blocking()*, *get_snake_lm_blocking()*, *get_standard_lm_blocking()*, *get_subblocks()*, *get_theta_blocking_cache()*, *get_theta_blocking_openmp()*

Needed modules

- *parallel_mod* (*n_procs()*, *rank_with_l1m0()*, *nthreads()*, rank(), *nlmbs_per_rank()*): This module contains the blocking information
- *useful* (*logwrite()*): library with several useful subroutines
- *output_data* (*log_file()*, *nlf()*): This module contains the parameters for output control
- *precision_mod*: This module controls the precision used in MagIC
- *logic* (*l_save_out()*): Module containing the logicals that control the run
- *lmmapping* (subblocks_mappings(), allocate_subblocks_mappings(), allocate_mappings(), mappings())
- *constants* (*one()*): module containing constants and parameters used in the code.
- *truncation* (*l_max()*, *minc()*, *lmp_max()*, *n_r_max()*, *lm_max()*, *nrp()*, *n_theta_max()*): This module defines the grid points and the truncation

Variables

- blocking/**nbsave** *[integer,private/parameter=16]*
- blocking/**lmstartb** (:) *[integer,allocatable/public]*
- blocking/**lm2** (:,:) *[integer,pointer/public]*
- blocking/**lmp2** (:,:) *[integer,pointer/public]*

- blocking/**sizerb** *[integer,public]*
- blocking/**lm2lms** (:) *[integer,pointer/public]*
- blocking/**lmp2l** (:) *[integer,pointer/public]*
- blocking/**lm22l** (:,:,:) *[integer,pointer/public]*
- blocking/**lm22m** (:,:,:) *[integer,pointer/public]*
- blocking/**lmp2lm** (:) *[integer,pointer/public]*
- blocking/**lm2lmp** (:) *[integer,pointer/public]*
- blocking/**get_theta_blocking** *[public]*
- blocking/**lo_sub_map** *[subblocks_mappings,target/public]*
- blocking/**lmp2lmps** (:) *[integer,pointer/public]*
- blocking/**nthetabs** *[integer,public]*
- blocking/**sn_sub_map** *[subblocks_mappings,target/public]*
- blocking/**lm2m** (:) *[integer,pointer/public]*
- blocking/**lm2l** (:) *[integer,pointer/public]*
- blocking/**lmp2lmpa** (:) *[integer,pointer/public]*
- blocking/**lm2mc** (:) *[integer,pointer/public]*
- blocking/**nlmbs** *[integer,public]*
- blocking/**sizethetab** *[integer,public]*
- blocking/**st_map** *[mappings,target/public]*
- blocking/**lo_map** *[mappings,target/public]*
- blocking/**nlmbs2** (:) *[integer,pointer/public]*
- blocking/**sizelmb** *[integer,public]*
- blocking/**sizethetabi** *[integer,private/parameter=284]*
- blocking/**lm22lm** (:,:,:) *[integer,pointer/public]*
- blocking/**nfs** *[integer,public]*
- blocking/**lmstopb** (:) *[integer,allocatable/public]*
- blocking/**l2lmas** (:) *[integer,pointer/public]*
- blocking/**cacheblock_size_in_b** *[integer,public]*
- blocking/**nbdown** *[integer,private/parameter=8]*
- blocking/**lm2lma** (:) *[integer,pointer/public]*
- blocking/**sizelmb2** (:,:) *[integer,pointer/public]*
- blocking/**st_sub_map** *[subblocks_mappings,target/public]*

Subroutines and functions

subroutine blocking/**initialize_blocking**()

Called from *magic*

Call to allocate_mappings(), *logwrite()*, *get_standard_lm_blocking()*, *get_snake_lm_blocking()*, *get_lorder_lm_blocking()*, allocate_subblocks_mappings(), *get_subblocks()*, *get_theta_blocking_cache()*, *get_theta_blocking_openmp()*

subroutine blocking/**get_subblocks**(*map*, *sub_map*)

Parameters

- **map** *[mappings,in]*
- **sub_map** *[subblocks_mappings,inout]*

Called from *initialize_blocking()*

subroutine blocking/**get_standard_lm_blocking**(*map*, *minc*)

Parameters

- **map** *[mappings,inout]*
- **minc** *[integer,in]*

Called from *initialize_blocking()*

subroutine blocking/**get_lorder_lm_blocking**(*map*, *minc*)

Parameters

- **map** *[mappings,inout]*
- **minc** *[integer,in]*

Called from *initialize_blocking()*

subroutine blocking/**get_snake_lm_blocking**(*map*, *minc*)

Parameters

- **map** *[mappings,inout]*
- **minc** *[integer,in]*

Called from *initialize_blocking()*

subroutine blocking/**get_theta_blocking_cache**(*n_theta_max*, *nrp*, *cacheblock_size_in_b*, *nthetabs*, *sizethetab*)

Parameters

- **n_theta_max** *[integer,in]*
- **nrp** *[integer,in]*
- **cacheblock_size_in_b** *[integer,in]*
- **nthetabs** *[integer,out]*
- **sizethetab** *[integer,out]*

Called from *initialize_blocking()*

subroutine blocking/**get_theta_blocking_openmp**(*n_theta_max*, *nthreads*, *nthetabs*, *sizethetab*)

This routine determines the number of theta blocks and the blocksize with respect to the number of threads.

Parameters

- **n_theta_max** *[integer,in]*
- **nthreads** *[integer,in]*
- **nthetabs** *[integer,out]*
- **sizethetab** *[integer,out]*

Called from *initialize_blocking()*

10.14.2 LMmapping.f90

Quick access

Routines `allocate_mappings(),allocate_subblocks_mappings()`

Types

- **type** `lmmapping/`**`unknown_type`**

  Type fields

  - `%` **`l_max`** *[integer]*
  - `%` **`lm2mc`** (*) *[integer,allocatable]*
  - `%` **`lm2`** (,) *[integer,allocatable]*
  - `%` **`lmp2l`** (*) *[integer,allocatable]*
  - `%` **`lmp2m`** (*) *[integer,allocatable]*
  - `%` **`lmp_max`** *[integer]*
  - `%` **`lmp2lm`** (*) *[integer,allocatable]*
  - `%` **`lmp2`** (,) *[integer,allocatable]*
  - `%` **`lm_max`** *[integer]*
  - `%` **`lmp2lmps`** (*) *[integer,allocatable]*
  - `%` **`lm2lms`** (*) *[integer,allocatable]*
  - `%` **`lm2lmp`** (*) *[integer,allocatable]*
  - `%` **`lm2m`** (*) *[integer,allocatable]*
  - `%` **`lm2l`** (*) *[integer,allocatable]*
  - `%` **`l2lmas`** (*) *[integer,allocatable]*
  - `%` **`lmp2lmpa`** (*) *[integer,allocatable]*
  - `%` **`lm2lma`** (*) *[integer,allocatable]*

- **type** `lmmapping/`**`unknown_type`**

  Type fields

- % **l_max** *[integer]*
- % **nlmbs** *[integer]*
- % **lm22lm** („,*) *[integer,allocatable]*
- % **lm22l** („,*) *[integer,allocatable]*
- % **lm22m** („,*) *[integer,allocatable]*
- % **sizelmb2** (,) *[integer,allocatable]*
- % **sizelmb2max** *[integer]*
- % **nlmbs2** (*) *[integer,allocatable]*

Variables

Subroutines and functions

subroutine lmmapping/**allocate_mappings** (*self*, *l_max*, *lm_max*, *lmp_max*)

Parameters

- **self** *[mappings]*
- **l_max** *[integer,in]*
- **lm_max** *[integer,in]*
- **lmp_max** *[integer,in]*

Called from *initialize_blocking()*

subroutine lmmapping/**allocate_subblocks_mappings** (*self*, *map*, *nlmbs*, *l_max*, *lmstartb*, *lmstopb*)

Parameters

- **self** *[subblocks_mappings]*
- **map** *[mappings,in]*
- **nlmbs** *[integer,in,]*
- **l_max** *[integer,in]*
- **lmstartb** (*nlmbs*) *[integer,in]*
- **lmstopb** (*nlmbs*) *[integer,in]*

Called from *initialize_blocking()*

10.15 IO: time series, radial profiles and spectra

10.15.1 output.f90

Quick access

Variables *n_cmb_setsmov*, *n_dt_cmb_sets*, *n_e_sets*, *n_spec*, *nbpotsets*, *nlogs*, *npvsets*, *nrms_sets*, *ntomovsets*, *ntormssets*, *ntosets*, *ntpotsets*, *nvpotsets*, *dteint*, *e_kin_pmean*, *e_kin_tmean*, *e_mag_pmean*, *e_mag_tmean*, *etot*,

etotold, *timenormlog*, *timenormrms*, *timepassedlog*, *timepassedrms*, *n_b_r_sets*, *n_t_r_sets*, *n_v_r_sets*, *dipcmbmean*, *dipmean*, *dlbmean*, *dlvcmean*, *dlvmean*, *dmbmean*, *dmvmean*, *dpvmean*, *dzvmean*, *elcmbmean*, *elmean*, *geosmean*, *lbdissmean*, *lvdissmean*, *rmmean*, *rolmean*

Routines *initialize_output()*, *output()*

Needed modules

- *horizontal_data* (*hdif_b()*, *dpl0eq()*, *dlh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *movie_data* (*movie_gather_frames_to_rank0()*)
- *useful* (*safeclose()*, *safeopen()*, *logwrite()*): library with several useful subroutines
- *fields* (*aj_lmloc()*, *aj_ic()*, *ddw_lmloc()*, *aj()*, *b_lmloc()*, *dj_lmloc()*, *ddj_ic_lmloc()*, *db_lmloc()*, *s_lmloc()*, *db_ic()*, *ds_lmloc()*, *b_ic()*, *ddb_lmloc()*, *ddb_ic_lmloc()*, *ddw()*, *ddj()*, *ddj_ic()*, *ddb()*, *db_ic_lmloc()*, *dw_lmloc()*, *dj_ic_lmloc()*, *ddb_ic()*, *dj()*, *db()*, *dj_ic()*, *aj_ic_lmloc()*, *dz()*, *z_lmloc()*, *dw()*, *omega_ma()*, *w_lmloc()*, *ds()*, *p_lmloc()*, *b()*, *b_ic_lmloc()*, *p()*, *s()*, *ddj_lmloc()*, *w()*, *omega_ic()*, *z()*, *dz_lmloc()*): This module contains the potential fields and their radial derivatives
- *rms* (*dtvrms()*, *zerorms()*, *dtbrms()*): This module contains the calculation of thr RMS force balance and induction terms.
- *lmloop_data* (*llmmag()*, *lm_per_rank()*, *lm_on_last_rank()*, *ulm()*, *llm()*, *ulmmag()*)
- *radial_spectra*
- *getdlm_mod* (*getdlm()*)
- *charmanip* (*dble2str()*): This module contains several useful routines to manipule character strings
- *magnetic_energy* (*get_e_mag()*)
- *store_pot_mod* (*storepot()*): This module contains the subroutines that can be used to write unformatted fortran files that contain the flow/magnetic field potentials (in both Chebyshev and spectral space)
- *integration* (*rint_r()*): Radial integration functions
- *graphout_mod* (*graphout_ic()*)
- *parallel_mod* (`rank()`): This module contains the blocking information
- *num_param* (*tscale()*): Module containing numerical and control parameters
- *out_movie_ic* (*store_movie_frame_ic()*)
- *physical_parameters* (*prmag()*, *lffac()*, *ek()*, *ktopv()*, *nvarcond()*, *opm()*): Module containing the physical parameters
- *radial_functions* (*drx()*, *r_cmb()*, *chebt_oc()*, *or2()*, *or1()*, *r()*, *r_icb()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *power* (*get_power()*)
- *logic* (*l_rmagspec()*, *l_power()*, *l_sric()*, *l_dt_cmb_field()*, *l_movie_ic()*, *l_non_rot()*, *l_dtb()*, *l_store_frame()*, *l_r_fieldt()*, *l_cmb_field()*, *l_average()*, *lverbose()*, *l_save_out()*, *l_pv()*, *l_cond_ic()*, *l_mag_lf()*, *l_anel()*, *l_perppar()*, *l_rms()*, *l_r_field()*, *l_mag()*): Module containing the logicals that control the run

- *precision_mod*: This module controls the precision used in MagIC
- *storecheckpoints*: This module contains several subroutines that can be used to store the rst_#.TAG files
- *kinetic_energy* (*get_e_kin()*, *get_u_square()*)
- *out_coeff* (*write_coeff_r()*, *write_bcmb()*): This module contains the subroutines that calculate the Bcmb files and the [B|V|T]_coeff_r files
- *omega* (*outomega()*): This module allows to compute the axisymmetric zonal flow versus the cylindrical radius s. By
- *blocking* (*lm2()*, *st_map()*, *lo_map()*): Module containing blocking information
- *constants* (*mass()*, *vol_oc()*, *surf_cmb()*, *vol_ic()*, *two()*): module containing constants and parameters used in the code.
- *truncation* (*n_r_ic_max()*, *l_max()*, *minc()*, *l_maxmag()*, *n_r_max()*, *lm_max()*, *n_r_maxmag()*): This module defines the grid points and the truncation
- *fieldslast* (*dbdt_iclast_lmloc()*, *dzdtlast_lo()*, *djdtlast_lmloc()*, *djdt_iclast_lmloc()*, *djdtlast()*, *dpdtlast()*, *djdt_iclast()*, *dpdtlast_lmloc()*, *dbdtlast()*, *dsdtlast()*, *dwdtlast_lmloc()*, *dsdtlast_lmloc()*, *dzdtlast()*, *dbdtlast_lmloc()*, *dbdt_iclast()*, *dwdtlast()*): This module contains time-derivaties array of the previous time-step They are needed in the time-stepping scheme.
- *outrot* (*write_rot()*)
- *out_movie* (*write_movie_frame()*)
- *output_data* (*n_cmbmov_file()*, *n_dt_cmb_file()*, *rst_file()*, *n_coeff_r()*, *tag()*, *nlf()*, *n_t_r_file()*, *n_coeff_r_max()*, *n_rst_file()*, *n_r_step()*, *n_par_file()*, *dt_cmb_file()*, *t_r_file()*, *cmb_file()*, *par_file()*, *n_v_r_file()*, *cmbmov_file()*, *n_cmb_file()*, *n_r_array()*, *l_max_cmb()*, *b_r_file()*, *v_r_file()*, *l_max_r()*, *n_b_r_file()*, *log_file()*): This module contains the parameters for output control
- *spectra* (*spectrum_temp()*, *spectrum_average()*, *spectrum_temp_average()*, *spectrum()*)
- *dtb_mod* (*get_dtblmfinish()*): This module contains magnetic field stretching and advection terms plus a separate omega-effect. It is used for movie output....
- *outmisc_mod* (*outmisc()*)
- *radial_data* (*nrstartmag()*, *n_r_cmb()*, *nrstart()*, *nrstopmag()*, *nrstop()*)
- *communications* (*gather_all_from_lo_to_rank0()*, *gt_ic()*, *gt_oc()*)
- *outpar_mod* (*outpar()*, *outperppar()*)
- *outto_mod* (*outto()*)
- *outpv3* (*outpv()*)
- *fields_average_mod* (*fields_average()*): This module is used when one wants to store time-averaged quantities

Variables

- output_mod/**dlvmean** *[real,private/save]*
- output_mod/**ntpotsets** *[integer,private]*
- output_mod/**etot** *[real,private]*

- output_mod/**n_v_r_sets** (:) *[integer,private/allocatable]*
- output_mod/**dmbmean** *[real,private/save]*
- output_mod/**ntosets** *[integer,private]*
- output_mod/**n_spec** *[integer,private]*
- output_mod/**n_b_r_sets** (:) *[integer,private/allocatable]*
- output_mod/**timepassedlog** *[real,private]*
- output_mod/**ntormssets** *[integer,private]*
- output_mod/**n_e_sets** *[integer,private]*
- output_mod/**nlogs** *[integer,private]*
- output_mod/**e_kin_pmean** *[real,private]*
- output_mod/**nbpotsets** *[integer,private]*
- output_mod/**dpvmean** *[real,private/save]*
- output_mod/**dipmean** *[real,private/save]*
- output_mod/**dteint** *[real,private]*
- output_mod/**nvpotsets** *[integer,private]*
- output_mod/**rolmean** *[real,private/save]*
- output_mod/**etotold** *[real,private]*
- output_mod/**dmvmean** *[real,private/save]*
- output_mod/**elmean** *[real,private/save]*
- output_mod/**lvdissmean** *[real,private/save]*
- output_mod/**n_dt_cmb_sets** *[integer,private]*
- output_mod/**e_mag_pmean** *[real,private]*
- output_mod/**elcmbmean** *[real,private/save]*
- output_mod/**timenormlog** *[real,private]*
- output_mod/**ntomovsets** *[integer,private]*
- output_mod/**geosmean** *[real,private/save]*
- output_mod/**timenormrms** *[real,private]*
- output_mod/**e_mag_tmean** *[real,private]*
- output_mod/**timepassedrms** *[real,private]*
- output_mod/**dzvmean** *[real,private/save]*
- output_mod/**dlvcmean** *[real,private/save]*
- output_mod/**n_cmb_setsmov** *[integer,private]*
- output_mod/**n_t_r_sets** (:) *[integer,private/allocatable]*
- output_mod/**npvsets** *[integer,private]*
- output_mod/**dlbmean** *[real,private/save]*
- output_mod/**dipcmbmean** *[real,private/save]*

- output_mod/**e_kin_tmean** *[real,private]*
- output_mod/**lbdissmean** *[real,private/save]*
- output_mod/**nrms_sets** *[integer,private]*
- output_mod/**rmmean** *[real,private/save]*

Subroutines and functions

subroutine output_mod/**initialize_output**()

Called from *magic*

subroutine output_mod/**output**(*time*, *dt*, *dtnew*, *n_time_step*, *l_stop_time*, *l_bpot*, *l_vpot*, *l_tpot*, *l_log*, *l_graph*, *lrmscalc*, *l_store*, *l_new_rst_file*, *l_spectrum*, *ltocalc*, *ltoframe*, *ltozwrite*, *l_frame*, *n_frame*, *l_cmb*, *n_cmb_sets*, *l_r*, *lorentz_torque_ic*, *lorentz_torque_ma*, *dbdt_at_cmb*, *hellmr*, *hel2lmr*, *helnalmr*, *helna2lmr*, *uhlmr*, *duhlmr*, *gradslmr*, *fconvlmr*, *fkinlmr*, *fvisclmr*, *fpoynlmr*, *freslmr*, *eperplmr*, *eparlmr*, *eperpaxilmr*, *eparaxilmr*)

This subroutine controls most of the output.

Parameters

- **time** *[real,in]*
- **dt** *[real,in]*
- **dtnew** *[real,in]*
- **n_time_step** *[integer,in]*
- **l_stop_time** *[logical,in]*
- **l_bpot** *[logical,in]*
- **l_vpot** *[logical,in]*
- **l_tpot** *[logical,in]*
- **l_log** *[logical,in]*
- **l_graph** *[logical,in]*
- **lrmscalc** *[logical,in]*
- **l_store** *[logical,in]*
- **l_new_rst_file** *[logical,in]*
- **l_spectrum** *[logical,in]*
- **ltocalc** *[logical,in]*
- **ltoframe** *[logical,in]*
- **ltozwrite** *[logical,inout]*
- **l_frame** *[logical,in]*
- **n_frame** *[integer,inout]*
- **l_cmb** *[logical,in]*
- **n_cmb_sets** *[integer,inout]*

- **l_r** *[logical,in]*
- **lorentz_torque_ic** *[real,in]*
- **lorentz_torque_ma** *[real,in]*
- **dbdt_at_cmb** (*) *[complex,in,pointer]*
- **hellmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **hel2lmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **helnalmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **helna2lmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **uhlmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **duhlmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **gradslmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **fconvlmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **fkinlmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **fvisclmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **fpoynlmr** (*l_maxmag*+1,*nrstopmag*-(*nrstartmag*)+1) *[real,in]*
- **freslmr** (*l_maxmag*+1,*nrstopmag*-(*nrstartmag*)+1) *[real,in]*
- **eperplmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **eparlmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **eperpaxilmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **eparaxilmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*

Called from *step_time()*

Call to *write_rot()*, *get_e_kin()*, *get_e_mag()*, *spectrum_average()*, *spectrum_temp_average()*, *fields_average()*, *get_power()*, *get_u_square()*, *outperppar()*, *getdlm()*, *outpar()*, *outmisc()*, *spectrum()*, *spectrum_temp()*, *outto()*, *get_dtblmfinish()*, *zerorms()*, *dtvrms()*, *dtbrms()*, *gather_all_from_lo_to_rank0()*, *movie_gather_frames_to_rank0()*, *graphout_ic()*, *rbrspec()*, *rbpspec()*, *store_movie_frame_ic()*, *logwrite()*, *write_movie_frame()*, *write_bcmb()*, *write_coeff_r()*, *rint_r()*, *safeopen()*, *safeclose()*, *storepot()*, *dble2str()*, *store()*, *outomega()*, *outpv()*

10.15.2 kinetic_energy.f90

Quick access

Routines *initialize_kinetic_energy()*, *get_e_kin()*, *get_u_square()*

Needed modules

- *horizontal_data* (*dlh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

- *precision_mod*: This module controls the precision used in MagIC
- *useful* (*cc2real()*): library with several useful subroutines
- *output_data* (*e_kin_file()*, *n_u_square_file()*, *tag()*, *n_e_kin_file()*, *u_square_file()*): This module contains the parameters for output control
- *logic* (*l_non_rot()*, *l_save_out()*): Module containing the logicals that control the run
- *lmloop_data* (*llm()*, *ulm()*)
- *integration* (*rint_r()*): Radial integration functions
- *parallel_mod*: This module contains the blocking information
- *communications* (*get_global_sum()*)
- *num_param* (*escale()*, *tscale()*): Module containing numerical and control parameters
- *physical_parameters* (*nvarcond()*, *prmag()*, *ek()*): Module containing the physical parameters
- *truncation* (*l_max()*, *n_r_max()*): This module defines the grid points and the truncation
- *blocking* (*lo_map()*, *st_map()*): Module containing blocking information
- *constants* (*two()*, *one()*, *four()*, *half()*, *vol_oc()*, *pi()*): module containing constants and parameters used in the code.
- *radial_functions* (*drx()*, *orho1()*, *orho2()*, *r_cmb()*, *chebt_oc()*, *or2()*, *or1()*, *r()*, *r_icb()*, *sigma()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

- kinetic_energy/**e_pa** (:) *[real,private/allocatable]*
- kinetic_energy/**e_ta** (:) *[real,private/allocatable]*
- kinetic_energy/**e_p_asa** (:) *[real,private/allocatable]*
- kinetic_energy/**e_t_asa** (:) *[real,private/allocatable]*

Subroutines and functions

subroutine kinetic_energy/**initialize_kinetic_energy**()

Called from *magic*

subroutine kinetic_energy/**get_e_kin**(*time*, *l_write*, *l_stop_time*, *n_e_sets*, *w*, *dw*, *z*, *e_p*, *e_t*, *e_p_as*, *e_t_as*[, *ekinr*])

Parameters

- **time** *[real,in]* :: Current time
- **l_write** *[logical,in]* :: Switch to write output
- **l_stop_time** *[logical,in]* :: Indicates when last time step of the run is reached for radial output
- **n_e_sets** *[integer,in]* :: Switch for time-average and to determine first time step
- **w** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **dw** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*

- **z** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **e_p** *[real,out]* :: poloidal energy
- **e_t** *[real,out]* :: toroidal energy
- **e_p_as** *[real,out]* :: axisymmetric poloidal energy
- **e_t_as** *[real,out]* :: axisymmetric toroidal energy

Options **ekinr** (*n_r_max*) *[real,out,optional]* :: Radial profile of kinetic energy

Called from *fields_average()*, *output()*

Call to *cc2real()*, *rint_r()*

subroutine kinetic_energy/**get_u_square** (*time*, *w*, *dw*, *z*, *rolr*, *dlr*, *dlrc*)

calculates square velocity = 1/2 Integral (v^2 dV) integration in theta,phi by summation of spherical harmonics integration in r by using Chebychef integrals

Write the different contributions in u_square.TAG file

Parameters

- **time** *[real,in]* :: Current time
- **w** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **dw** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **z** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **rolr** (*n_r_max*) *[real,out]* :: local Rossby number
- **dlr** (*n_r_max*) *[real,out]* :: Length scale
- **dlrc** (*n_r_max*) *[real,out]* :: Convective length scale

Called from *output()*

Call to *cc2real()*, *rint_r()*

10.15.3 magnetic_energy.f90

Quick access

Variables *e_dipa*, *e_p_asa*, *e_pa*, *e_t_asa*, *e_ta*

Routines *initialize_magnetic_energy()*, *get_e_mag()*

Needed modules

- *horizontal_data* (*dlh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *precision_mod*: This module controls the precision used in MagIC
- *bext* (*n_imp()*, *rrmp()*): Module containing the external field parameters
- *useful* (*cc22real()*, *cc2real()*): library with several useful subroutines
- *movie_data* (*moviedipstrengthgeo()*, *moviedipstrength()*, *moviedipcolat()*, *moviediplon()*)

- *output_data* (*n_e_mag_ic_file()*, *e_mag_oc_file()*, *tag()*, *n_dipole_file()*, *n_e_mag_oc_file()*, *e_mag_ic_file()*, *dipole_file()*): This module contains the parameters for output control
- *logic* (*l_save_out()*, *l_cond_ic()*, *l_mag()*, *l_mag_lf()*): Module containing the logicals that control the run
- *lmloop_data* (*llmmag()*, *ulmmag()*)
- *radial_data* (*n_r_cmb()*)
- *parallel_mod*: This module contains the blocking information
- *integration* (*rint_r()*, *rintic()*): Radial integration functions
- *num_param* (*escale()*, *tscale()*): Module containing numerical and control parameters
- *physical_parameters* (*ktopb()*, *lffac()*, *kbotb()*): Module containing the physical parameters
- *truncation* (*n_r_ic_max()*, *n_r_max()*, *n_r_maxmag()*, *n_r_ic_maxmag()*): This module defines the grid points and the truncation
- *blocking* (*lo_map()*, *lmstartb()*, *st_map()*, *lmstopb()*): Module containing blocking information
- *constants* (*two()*, *one()*, *four()*, *zero()*, *half()*, *pi()*): module containing constants and parameters used in the code.
- *radial_functions* (*drx()*, *orho1()*, *r_cmb()*, *chebt_oc()*, *or2()*, *r_ic()*, *r()*, *dr_fac_ic()*, *r_icb()*, *chebt_ic()*, *sigma()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

- magnetic_energy/**e_pa** (:) *[real,private/allocatable]*
- magnetic_energy/**e_ta** (:) *[real,private/allocatable]*
- magnetic_energy/**e_dipa** (:) *[real,private/allocatable]*
- magnetic_energy/**e_p_asa** (:) *[real,private/allocatable]*
- magnetic_energy/**e_t_asa** (:) *[real,private/allocatable]*

Subroutines and functions

subroutine magnetic_energy/**initialize_magnetic_energy**()

Called from *magic*

subroutine magnetic_energy/**get_e_mag**(*time*, *l_write*, *l_stop_time*, *n_e_sets*, *b*, *db*, *aj*, *b_ic*, *db_ic*, *aj_ic*, *e_p*, *e_t*, *e_p_as*, *e_t_as*, *e_p_ic*, *e_t_ic*, *e_p_as_ic*, *e_t_as_ic*, *e_p_os*, *e_p_as_os*, *e_cmb*, *dip*, *dipcmb*, *elsanel*)

calculates magnetic energy = 1/2 Integral(B^2 dV) integration in theta,phi by summation over harmonic coeffs. integration in r by Chebycheff integrals

Parameters

- **time** *[real,in]* :: Current time
- **l_write** *[logical,in]* :: Switch to write output

- **l_stop_time** *[logical,in]* :: Indicates when last time step of the run is reached for radial output
- **n_e_sets** *[integer,in]* :: Switch for time-average and to determine first time step
- **b** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,in]*
- **db** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,in]*
- **aj** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,in]*
- **b_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,in]*
- **db_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,in]*
- **aj_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,in]*
- **e_p** *[real,out]* :: Volume averaged poloidal magnetic energy
- **e_t** *[real,out]* :: Volume averaged toroidal magnetic energy
- **e_p_as** *[real,out]* :: Volume averaged axisymmetric poloidal magnetic energy
- **e_t_as** *[real,out]* :: Volume averaged axisymmetric toroidal magnetic energy
- **e_p_ic** *[real,out]* :: IC poloidal magnetic energy
- **e_t_ic** *[real,out]* :: IC toroidal magnetic energy
- **e_p_as_ic** *[real,out]* :: IC axisymmetric poloidal magnetic energy
- **e_t_as_ic** *[real,out]* :: IC axisymmetric toroidal magnetic energy
- **e_p_os** *[real,out]* :: Outside poloidal magnetic energy
- **e_p_as_os** *[real,out]* :: Outside axisymmetric poloidal magnetic energy
- **e_cmb** *[real,out]* :: Magnetic energy at the CMB
- **dip** *[real,out]* :: Relative magnetic energy of axial dipole
- **dipcmb** *[real,out]* :: Relative magnetic energy of axial dipole at the CMB
- **elsanel** *[real,out]* :: Radially averaged Elsasser number

Called from *fields_average()*, *output()*

Call to *cc2real()*, *rint_r()*, *cc22real()*, *rintic()*

10.15.4 getDlm.f90

Quick access

Routines *getdlm()*

Needed modules

- *horizontal_data* (*dlh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *precision_mod*: This module controls the precision used in MagIC
- *useful* (*cc22real()*, *cc2real()*): library with several useful subroutines
- *lmloop_data* (*llm()*, *ulm()*)

- *integration* (*rint_r()*): Radial integration functions
- *parallel_mod*: This module contains the blocking information
- *num_param* (*escale()*): Module containing numerical and control parameters
- *truncation* (*l_max()*, *minc()*, *n_r_max()*, *m_max()*): This module defines the grid points and the truncation
- *blocking* (*lo_map()*, *st_map()*): Module containing blocking information
- *constants* (*pi()*, *half()*): module containing constants and parameters used in the code.
- *radial_functions* (*orho1()*, *drx()*, *or2()*, *chebt_oc()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

Subroutines and functions

subroutine getdlm_mod/**getdlm**(*w*, *dw*, *z*, *dl*, *dlr*, *dm*, *dlc*, *dlrc*, *switch_bn*)

calculates energy = 1/2 Integral(B^2 dV) integration in theta,phi by summation over harmonic coeffs. integration in r by Chebycheff integrals

Output: enbp: Total poloidal enbt: Total toroidal apome: Axisym. poloidal atome: Axisym. toroidal

Parameters

- **w** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **dw** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **z** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **dl** *[real,out]*
- **dlr** (*n_r_max*) *[real,out]*
- **dm** *[real,out]*
- **dlc** *[real,out]*
- **dlrc** (*n_r_max*) *[real,out]*
- **switch_bn** *[character,in]*

Called from *output()*

Call to *cc2real()*, *rint_r()*

10.15.5 outMisc.f90

Quick access

Routines *outmisc()*

Needed modules

- *horizontal_data* (*gauss()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *precision_mod*: This module controls the precision used in MagIC
- *egeos_mod* (*getegeos()*)
- *useful* (*cc2real()*): library with several useful subroutines
- *output_data* (*misc_file()*, *tag()*, *n_misc_file()*): This module contains the parameters for output control
- *logic* (*l_heat()*, *l_par()*, *l_save_out()*, *l_anelastic_liquid()*, *l_hel()*): Module containing the logicals that control the run
- *legendre_spec_to_grid* (*lmas2pt()*)
- *integration* (*rint()*, *rint_r()*): Radial integration functions
- *radial_data* (*n_r_cmb()*, *nrstart()*, *n_r_icb()*, *nrstop()*)
- *parallel_mod*: This module contains the blocking information
- *num_param* (*lscale()*): Module containing numerical and control parameters
- *lmloop_data* (*llm()*, *ulm()*)
- *physical_parameters* (*epss()*): Module containing the physical parameters
- *truncation* (*lm_max()*, *l_max()*, *n_r_max()*): This module defines the grid points and the truncation
- *blocking* (*lo_map()*, *nthetabs()*, *nfs()*, *sizethetab()*): Module containing blocking information
- *constants* (*four()*, *osq4pi()*, *two()*, *one()*, *sq4pi()*, *vol_oc()*, *pi()*): module containing constants and parameters used in the code.
- *radial_functions* (*botcond()*, *rho0()*, *kappa()*, *r_cmb()*, *chebt_oc()*, *dr_fac()*, *topcond()*, *temp0()*, *r_icb()*, *r()*, *dtemp0()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

Subroutines and functions

subroutine outmisc_mod/**outmisc** (*timescaled*, *hellmr*, *hel2lmr*, *helnalmr*, *helna2lmr*, *nlogs*, *w*, *dw*, *ddw*, *z*, *dz*, *s*, *ds*, *geos*, *dpflow*, *dzflow*)

Parameters

- **timescaled** *[real,in]*
- **hellmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **hel2lmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **helnalmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **helna2lmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **nlogs** *[integer,in]*
- **w** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*

- **dw** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **ddw** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **z** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **dz** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **s** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **ds** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **geos** *[real,out]*
- **dpflow** *[real,out]*
- **dzflow** *[real,out]*

Called from *output()*

Call to *lmas2pt()*, *rint()*, *getegeos()*

10.15.6 outRot.f90

Quick access

Variables *get_viscous_torque*

Routines *sendvals_to_rank0()*, *get_lorentz_torque()*, *write_rot()*, *get_viscous_torque_complex()*, *get_viscous_torque_real()*, *get_angular_moment()*

Needed modules

- *horizontal_data* (*gauss()*, *costheta()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *precision_mod*: This module controls the precision used in MagIC
- *output_data* (*angular_file()*, *n_angular_file()*, *srma_file()*, *sric_file()*, *rot_file()*, *tag()*, *n_srma_file()*, *n_rot_file()*, *n_sric_file()*): This module contains the parameters for output control
- *logic* (*l_rot_ic()*, *l_drift()*, *l_iner()*, *l_save_out()*, *l_srma()*, *l_mag_lf()*, *l_rot_ma()*, *l_am()*, *l_sric()*, *l_mag()*): Module containing the logicals that control the run
- *lmloop_data* (*ulmmag()*, *llmmag()*, *llm()*, *ulm()*)
- *radial_data* (*n_r_cmb()*, *n_r_icb()*)
- *parallel_mod*: This module contains the blocking information
- *integration* (*rint()*, *rint_r()*): Radial integration functions
- *num_param* (*tscale()*, *lscale()*, *vscale()*): Module containing numerical and control parameters
- *physical_parameters* (*kbotv()*, *ktopv()*): Module containing the physical parameters
- *truncation* (*nrp()*, *minc()*, *n_phi_max()*, *n_r_max()*, *n_r_maxmag()*): This module defines the grid points and the truncation
- *grenoble* (*lgrenoble()*, *bic()*): This module contains all variables for the case of an imposed IC dipole

- *blocking* (*lo_map()*, *lmstartb()*, *st_map()*, *lm2()*, *lmstopb()*): Module containing blocking information
- *constants* (*c_moi_oc()*, *y11_norm()*, *third()*, *y10_norm()*, *two()*, *four()*, *zero()*, *c_moi_ic()*, *half()*, *c_moi_ma()*, *pi()*): module containing constants and parameters used in the code.
- *radial_functions* (*r_icb()*, *r_cmb()*, *r()*, *chebt_oc()*, *drx()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

- outrot/**get_viscous_torque** *[public]*

Subroutines and functions

subroutine outrot/**write_rot**(*time*, *dt*, *ekinic*, *ekinma*, *w*, *z*, *dz*, *b*, *omega_ic*, *omega_ma*, *lorentz_torque_ic*, *lorentz_torque_ma*)

Parameters

- **time** *[real,in]*
- **dt** *[real,in]*
- **ekinic** *[real,out]*
- **ekinma** *[real,out]*
- **w** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **z** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **dz** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **b** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,in]*
- **omega_ic** *[real,in]*
- **omega_ma** *[real,in]*
- **lorentz_torque_ic** *[real,in]*
- **lorentz_torque_ma** *[real,in]*

Called from *output()*

Call to *sendvals_to_rank0()*, *get_angular_moment()*

subroutine outrot/**get_viscous_torque_real**(*viscous_torque*, *z10*, *dz10*, *r*)

Purpose of this subroutine is to calculate the viscous torque on mantle or inner core respectively. NOTE: sign is wrong for torque on mantle!

Parameters

- **viscous_torque** *[real,out]*
- **z10** *[real,in]* :: z10 coefficient and its radial deriv.
- **dz10** *[real,in]*
- **r** *[real,in]* :: radius (ICB or CMB)

subroutine outrot/**get_viscous_torque_complex**(*viscous_torque*, *z10*, *dz10*, *r*)

Purpose of this subroutine is to calculate the viscous torque on mantle or inner core respectively. NOTE: sign is wrong for torque on mantle!

Parameters

- **viscous_torque** *[real,out]*
- **z10** *[complex,in]* :: z10 coefficient and its radial deriv.
- **dz10** *[complex,in]*
- **r** *[real,in]* :: radius (ICB or CMB)

subroutine outrot/**get_lorentz_torque**(*lorentz_torque*, *nthetastart*, *sizethetab*, *br*, *bp*, *nr*)

Parameters

- **lorentz_torque** *[real,inout]* :: lorentz_torque for theta(1:n_theta)
- **nthetastart** *[integer,in]* :: first number of theta in block
- **sizethetab** *[integer,in]* :: size of theta bloching
- **br** (*nrp*,*) *[real,in]* :: array containing
- **bp** (*nrp*,*) *[real,in]* :: array containing
- **nr** *[integer,in]*

Called from *do_iteration_thetablocking_openmp()*, do_iteration_thetablocking_seq(), do_iteration_thetablocking_shtns()

subroutine outrot/**get_angular_moment**(*z10*, *z11*, *omega_ic*, *omega_ma*, *angular_moment_oc*, *angular_moment_ic*, *angular_moment_ma*)

Parameters

- **z10** (*n_r_max*) *[complex,in]*
- **z11** (*n_r_max*) *[complex,in]*
- **omega_ic** *[real,in]*
- **omega_ma** *[real,in]*
- **angular_moment_oc** (*) *[real,out]*
- **angular_moment_ic** (*) *[real,out]*
- **angular_moment_ma** (*) *[real,out]*

Called from *updatez()*, *write_rot()*

Call to *rint_r()*

subroutine outrot/**sendvals_to_rank0**(*field*, *n_r*, *lm_vals*, *vals_on_rank0*)

Parameters

- **field** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **n_r** *[integer,in]*
- **lm_vals** (*) *[integer,in]*
- **vals_on_rank0** (*) *[complex,out]*

Called from *write_rot()*

10.15.7 outPar.f90

Quick access

Variables *dlvcmeanr*, *dlvmeanr*, *dlvu2cmeanr*, *dlvu2meanr*, *duhmeanr*, *eparaximeanr*, *eparmeanr*, *eperpaximeanr*, *eperpmeanr*, *fcondmeanr*, *fconvmeanr*, *fkinmeanr*, *fpoynmeanr*, *fresmeanr*, *fviscmeanr*, *gradt2meanr*, *mvar*, *rmmeanr*, *rolmeanr*, *rolmeanru2*, *smeanr*, *svar*, *uhmeanr*

Routines *initialize_outpar_mod()*, *outperppar()*, *outpar()*

Needed modules

- *horizontal_data* (*gauss()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *precision_mod*: This module controls the precision used in MagIC
- *output_data* (*tag()*, *perppar_file()*, *n_perppar_file()*): This module contains the parameters for output control
- *useful* (*cc2real()*): library with several useful subroutines
- *fields* (*ds_rloc()*, *s_rloc()*): This module contains the potential fields and their radial derivatives
- *num_param* (*tscale()*): Module containing numerical and control parameters
- *legendre_spec_to_grid* (*lmas2pt()*)
- *integration* (*rint()*): Radial integration functions
- *radial_data* (*nrstartmag()*, *nrstart()*, *nrstopmag()*, *n_r_icb()*, *nrstop()*)
- *parallel_mod*: This module contains the blocking information
- *logic* (*l_mag_nl()*, *l_save_out()*, *l_anel()*, *l_perppar()*, *l_fluxprofs()*, *l_viscbccalc()*): Module containing the logicals that control the run
- *physical_parameters* (*prmag()*, *kbots()*, *ek()*, *opr()*, *vischeatfac()*, *ktops()*, *ohmlossfac()*): Module containing the physical parameters
- *radial_functions* (*dr_fac()*, *kappa()*, *chebt_oc()*, *rho0()*, *or2()*, *temp0()*, *r()*, *sigma()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *blocking* (*lm2m()*, *nthetabs()*, *nfs()*, *sizethetab()*): Module containing blocking information
- *constants* (*four()*, *osq4pi()*, *two()*, *sq4pi()*, *mass()*, *half()*, *pi()*): module containing constants and parameters used in the code.
- *truncation* (*lm_max()*, *l_max()*, *l_maxmag()*, *n_r_max()*, *n_r_maxmag()*): This module defines the grid points and the truncation

Variables

- outpar_mod/**fconvmeanr** (:) *[real,private/allocatable]*
- outpar_mod/**eparmeanr** (:) *[real,private/allocatable]*
- outpar_mod/**dlvcmeanr** (:) *[real,private/allocatable]*

- outpar_mod/**dlvu2cmeanr** (:) *[real,private/allocatable]*
- outpar_mod/**fcondmeanr** (:) *[real,private/allocatable]*
- outpar_mod/**rolmeanru2** (:) *[real,private/allocatable]*
- outpar_mod/**eparaximeanr** (:) *[real,private/allocatable]*
- outpar_mod/**fpoynmeanr** (:) *[real,private/allocatable]*
- outpar_mod/**rolmeanr** (:) *[real,private/allocatable]*
- outpar_mod/**fresmeanr** (:) *[real,private/allocatable]*
- outpar_mod/**fviscmeanr** (:) *[real,private/allocatable]*
- outpar_mod/**duhmeanr** (:) *[real,private/allocatable]*
- outpar_mod/**rmmeanr** (:) *[real,private/allocatable]*
- outpar_mod/**mvar** (:) *[real,private/allocatable]*
- outpar_mod/**smeanr** (:) *[real,private/allocatable]*
- outpar_mod/**gradt2meanr** (:) *[real,private/allocatable]*
- outpar_mod/**dlvu2meanr** (:) *[real,private/allocatable]*
- outpar_mod/**fkinmeanr** (:) *[real,private/allocatable]*
- outpar_mod/**uhmeanr** (:) *[real,private/allocatable]*
- outpar_mod/**svar** (:) *[real,private/allocatable]*
- outpar_mod/**eperpaximeanr** (:) *[real,private/allocatable]*
- outpar_mod/**dlvmeanr** (:) *[real,private/allocatable]*
- outpar_mod/**eperpmeanr** (:) *[real,private/allocatable]*

Subroutines and functions

subroutine outpar_mod/**initialize_outpar_mod**()

Called from *magic*

subroutine outpar_mod/**outpar**(*timepassed*, *timenorm*, *nlogs*, *l_stop_time*, *ekinr*, *rolru2*, *dlvr*, *dlvrc*, *dlvru2*, *dlvru2c*, *uhlmr*, *duhlmr*, *gradslmr*, *fconvlmr*, *fkinlmr*, *fvisclmr*, *fpoynlmr*, *freslmr*, *rmr*)

Parameters

- **timepassed** *[real,in]*
- **timenorm** *[real,in]*
- **nlogs** *[integer,in]*
- **l_stop_time** *[logical,in]*
- **ekinr** (*n_r_max*) *[real,in]* :: kinetic energy w radius
- **rolru2** (*n_r_max*) *[real,in]*
- **dlvr** (*n_r_max*) *[real,in]*
- **dlvrc** (*n_r_max*) *[real,in]*
- **dlvru2** (*n_r_max*) *[real,in]*

- **dlvru2c** (*n_r_max*) *[real,in]*
- **uhlmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **duhlmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **gradslmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **fconvlmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **fkinlmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **fvisclmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **fpoynlmr** (*l_maxmag*+1,*nrstopmag*-(*nrstartmag*)+1) *[real,in]*
- **freslmr** (*l_maxmag*+1,*nrstopmag*-(*nrstartmag*)+1) *[real,in]*
- **rmr** (*n_r_max*) *[real,out]*

Called from *output()*

Call to *cc2real()*, *lmas2pt()*

subroutine outpar_mod/**outperppar**(*time*, *timepassed*, *timenorm*, *l_stop_time*, *eperplmr*, *eparlmr*, *eperpaxilmr*, *eparaxilmr*)

Parameters

- **time** *[real,in]*
- **timepassed** *[real,in]*
- **timenorm** *[real,in]*
- **l_stop_time** *[logical,in]*
- **eperplmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **eparlmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **eperpaxilmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
- **eparaxilmr** (*l_max*+1,*nrstop*-(*nrstart*)+1) *[real,in]*

Called from *output()*

Call to *lmas2pt()*, *rint()*

10.15.8 power.f90

Quick access

Variables *buomeanr*, *curlu2meanr*, *ohmdissr*

Routines *initialize_output_power()*, *get_power()*

Needed modules

- *horizontal_data* (*dlh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *precision_mod*: This module controls the precision used in MagIC
- *useful* (*cc22real()*, *cc2real()*): library with several useful subroutines

- *outrot* (*get_viscous_torque()*)
- *output_data* (*tag()*, *n_power_file()*, *power_file()*): This module contains the parameters for output control
- *logic* (*l_rot_ic()*, *l_mag()*, *l_save_out()*, *l_srma()*, *l_rot_ma()*, *l_heat()*, *l_sric()*, *l_cond_ic()*, *l_conv()*): Module containing the logicals that control the run
- *lmloop_data* (*ulmmag()*, *llmmag()*, *llm()*, *ulm()*)
- *radial_data* (*n_r_cmb()*, *n_r_icb()*)
- *parallel_mod*: This module contains the blocking information
- *integration* (*rint_r()*, *rintic()*): Radial integration functions
- *num_param* (*escale()*, *tscale()*): Module containing numerical and control parameters
- *physical_parameters* (*kbotv()*, *ktopv()*, *lffac()*, *opm()*): Module containing the physical parameters
- *truncation* (*n_r_ic_max()*, *n_r_maxmag()*, *n_r_ic_maxmag()*, *n_r_max()*): This module defines the grid points and the truncation
- *blocking* (*lo_map()*, *lmstartb()*, *st_map()*, *lmstopb()*): Module containing blocking information
- *constants* (*half()*, *two()*, *one()*): module containing constants and parameters used in the code.
- *radial_functions* (*drx()*, *o_r_ic2()*, *r_ic()*, *r_cmb()*, *chebt_oc()*, *or2()*, *rgrav()*, *r()*, *o_r_ic()*, *dr_fac_ic()*, *r_icb()*, *chebt_ic()*, *lambda()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

- power/**ohmdissr** (:) *[real,private/allocatable]*
- power/**curlu2meanr** (:) *[real,private/allocatable]*
- power/**buomeanr** (:) *[real,private/allocatable]*

Subroutines and functions

subroutine power/**initialize_output_power**()

Called from *magic*

subroutine power/**get_power**(*time*, *timepassed*, *timenorm*, *l_stop_time*, *omega_ic*, *omega_ma*, *lorentz_torque_ic*, *lorentz_torque_ma*, *w*, *ddw*, *z*, *dz*, *s*, *b*, *ddb*, *aj*, *dj*, *db_ic*, *ddb_ic*, *aj_ic*, *dj_ic*, *viscdiss*, *ohmdiss*)

This subroutine calculates power and dissipation of the core/mantle system. Energy input into the outer core is by buoyancy and possibly viscous accelarations at the boundaries if the rotation rates of inner core or mantle are prescribed and kept fixed. The losses are due to Ohmic and viscous dissipation. If inner core and mantel are allowed to change their rotation rates due to viscous forces this power is not lost from the system and has to be respected.

The output is written into a file power.TAG.

Parameters

- **time** *[real,in]*

- **timepassed** *[real,in]*
- **timenorm** *[real,in]*
- **l_stop_time** *[logical,in]*
- **omega_ic** *[real,in]*
- **omega_ma** *[real,in]*
- **lorentz_torque_ic** *[real,in]*
- **lorentz_torque_ma** *[real,in]*
- **w** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **ddw** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **z** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **dz** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **s** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **b** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,in]*
- **ddb** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,in]*
- **aj** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,in]*
- **dj** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,in]*
- **db_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,in]*
- **ddb_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,in]*
- **aj_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,in]*
- **dj_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,in]*
- **viscdiss** *[real,out]*
- **ohmdiss** *[real,out]*

Called from `output()`

Call to `cc2real()`, `cc22real()`, `rint_r()`, `rintic()`

10.15.9 `spectra.f90`

Quick access

Variables `dt_icb2_ave`, `dt_icb_ave`, `e_cmb2_l_ave`, `e_cmb2_m_ave`, `e_cmb_l_ave`, `e_cmb_m_ave`, `e_p2_l_ave`, `e_p2_m_ave`, `e_p_l_ave`, `e_p_m_ave`, `e_t2_l_ave`, `e_t2_m_ave`, `e_t_l_ave`, `e_t_m_ave`, `ek_p2_l_ave`, `ek_p2_m_ave`, `ek_p_l_ave`, `ek_p_m_ave`, `ek_t2_l_ave`, `ek_t2_m_ave`, `ek_t_l_ave`, `ek_t_m_ave`, `t2_ave`, `t_ave`, `t_icb2_ave`, `t_icb_ave`

Routines `initialize_spectra()`, `spectrum_average()`, `spectrum_temp_average()`, `spectrum_temp()`, `spectrum()`, `get_standard_deviation()`

Needed modules

- *horizontal_data* (*dlh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *precision_mod*: This module controls the precision used in MagIC
- *useful* (*safeclose()*, *safeopen()*, *cc22real()*, *cc2real()*): library with several useful subroutines
- *output_data* (*n_kin_spec_file()*, *n_u2_spec_file()*, *tag()*, *nlf()*, *n_mag_spec_file()*, *log_file()*): This module contains the parameters for output control
- *logic* (*l_heat()*, *l_cond_ic()*, *l_mag()*, *l_anel()*): Module containing the logicals that control the run
- *lmloop_data* (*ulmmag()*, *llmmag()*, *llm()*, *ulm()*)
- *radial_data* (*n_r_cmb()*, *n_r_icb()*)
- *parallel_mod*: This module contains the blocking information
- *integration* (*rint()*, *rint_r()*, *rintic()*): Radial integration functions
- *num_param* (*escale()*, *tscale()*): Module containing numerical and control parameters
- *physical_parameters* (*lffac()*): Module containing the physical parameters
- *truncation* (*n_r_ic_max()*, *l_max()*, *n_r_ic_maxmag()*, *minc()*, *n_r_max()*, *n_r_maxmag()*): This module defines the grid points and the truncation
- *blocking* (*lo_map()*, *st_map()*): Module containing blocking information
- *constants* (*one()*, *four()*, *vol_oc()*, *pi()*, *half()*): module containing constants and parameters used in the code.
- *radial_functions* (*drx()*, *orho1()*, *orho2()*, *chebt_oc()*, *dr_fac()*, *or2()*, *r_ic()*, *r()*, *dr_fac_ic()*, *r_icb()*, *chebt_ic()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

- spectra/**e_cmb2_m_ave** (:) *[real,private/allocatable]*
- spectra/**ek_p_l_ave** (:) *[real,private/allocatable]*
- spectra/**e_t2_l_ave** (:) *[real,private/allocatable]*
- spectra/**ek_t_l_ave** (:) *[real,private/allocatable]*
- spectra/**t_icb2_ave** (:) *[real,private/allocatable]*
- spectra/**e_p_l_ave** (:) *[real,private/allocatable]*
- spectra/**e_cmb2_l_ave** (:) *[real,private/allocatable]*
- spectra/**e_cmb_l_ave** (:) *[real,private/allocatable]*
- spectra/**ek_p_m_ave** (:) *[real,private/allocatable]*
- spectra/**e_p2_l_ave** (:) *[real,private/allocatable]*
- spectra/**t_icb_ave** (:) *[real,private/allocatable]*
- spectra/**e_p_m_ave** (:) *[real,private/allocatable]*

- spectra/**dt_icb2_ave** (:) *[real,private/allocatable]*
- spectra/**e_t_m_ave** (:) *[real,private/allocatable]*
- spectra/**t_ave** (:) *[real,private/allocatable]*
- spectra/**e_p2_m_ave** (:) *[real,private/allocatable]*
- spectra/**ek_p2_m_ave** (:) *[real,private/allocatable]*
- spectra/**e_t2_m_ave** (:) *[real,private/allocatable]*
- spectra/**dt_icb_ave** (:) *[real,private/allocatable]*
- spectra/**ek_t_m_ave** (:) *[real,private/allocatable]*
- spectra/**t2_ave** (:) *[real,private/allocatable]*
- spectra/**ek_p2_l_ave** (:) *[real,private/allocatable]*
- spectra/**ek_t2_m_ave** (:) *[real,private/allocatable]*
- spectra/**e_t_l_ave** (:) *[real,private/allocatable]*
- spectra/**e_cmb_m_ave** (:) *[real,private/allocatable]*
- spectra/**ek_t2_l_ave** (:) *[real,private/allocatable]*

Subroutines and functions

subroutine spectra/**initialize_spectra**()

Called from *magic*

subroutine spectra/**spectrum_average**(*n_time_ave*, *l_stop_time*, *time_passed*, *time_norm*, *b*, *aj*, *db*, *bv*)

Parameters

- **n_time_ave** *[integer,in]*
- **l_stop_time** *[logical,in]*
- **time_passed** *[real,in]*
- **time_norm** *[real,in]*
- **b** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **aj** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **db** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **bv** *[character,in]*

Called from *output()*

Call to *cc2real()*, *rint_r()*, *get_standard_deviation()*, *safeopen()*, *safeclose()*

function spectra/**get_standard_deviation**(*dt_norm*, *mean*, *sum_of_squares*)

Parameters

- **dt_norm** *[real,in]*
- **mean** *[real,in]*
- **sum_of_squares** *[real,in]*

Return stdev *[real]*

Called from *spectrum_average()*, *spectrum_temp_average()*

subroutine spectra/**spectrum**(*time*, *n_spec*, *w*, *dw*, *z*, *b*, *db*, *aj*, *b_ic*, *db_ic*, *aj_ic*)

calculates magnetic energy = 1/2 Integral(B^2 dV) integration in theta,phi by summation over harmonic coeffs. integration in r by Chebycheff integrals

Output: enbp: Total poloidal enbt: Total toroidal apome: Axisym. poloidal atome: Axisym. toroidal

Parameters

- **time** *[real,in]*
- **n_spec** *[integer,in]* :: number of spectrum/call, file
- **w** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **dw** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **z** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **b** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,in]*
- **db** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,in]*
- **aj** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,in]*
- **b_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,in]*
- **db_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,in]*
- **aj_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,in]*

Called from *fields_average()*, *output()*

Call to *cc2real()*, *rint_r()*, *cc22real()*, *rintic()*

subroutine spectra/**spectrum_temp_average**(*n_time_ave*, *l_stop_time*, *time_passed*, *time_norm*, *s*, *ds*)

Parameters

- **n_time_ave** *[integer,in]*
- **l_stop_time** *[logical,in]*
- **time_passed** *[real,in]*
- **time_norm** *[real,in]*
- **s** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **ds** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*

Called from *output()*

Call to *cc2real()*, *rint()*, *get_standard_deviation()*, *safeopen()*, *safeclose()*

subroutine spectra/**spectrum_temp**(*time*, *n_spec*, *s*, *ds*)

calculates spectra of temperature and composition

Parameters

- **time** *[real,in]*

- **n_spec** *[integer,in]* :: number of spectrum/call, file
- **s** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **ds** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*

Called from *fields_average()*, *output()*

Call to *cc2real()*, *rint()*

10.16 IO: graphic files, movie files, coeff files and potential files

10.16.1 out_graph_file.f90

Quick access

Routines *graphout_ic()*, *graph_write_mpi()*, *graph_write()*, *graphout_header()*, *graphout_mpi_header()*, *graphout_mpi()*, *graphout()*

Needed modules

- *horizontal_data* (*dplm()*, *plm()*, *dlh()*, *theta_ord()*, *o_sin_theta()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *precision_mod*: This module controls the precision used in MagIC
- *fft*: This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5
- *output_data* (*n_graph_file()*, *runid()*, *graph_mpi_fh()*): This module contains the parameters for output control
- *logic* (*l_pressgraph()*, *l_cond_ic()*, *l_mag()*): Module containing the logicals that control the run
- *legendre_spec_to_grid* (*legtf()*)
- *radial_data* (*n_r_icb()*)
- *parallel_mod*: This module contains the blocking information
- *num_param* (*vscale()*): Module containing numerical and control parameters
- *physical_parameters* (*pr()*, *prmag()*, *ek()*, *sigma_ratio()*, *radratio()*, *ra()*): Module containing the physical parameters
- *truncation* (*n_r_ic_max()*, *lm_maxmag()*, *nrp()*, *n_r_ic_maxmag()*, *minc()*, *n_phi_max()*, *l_max()*, *n_r_max()*, *lm_max()*, *n_theta_max()*, *n_r_maxmag()*, *n_phi_tot()*): This module defines the grid points and the truncation
- *leg_helper_mod* (*legprep_ic()*)
- *blocking* (*nthetabs()*, *nfs()*, *sizethetab()*): Module containing blocking information
- *radial_functions* (*orho1()*, *r_cmb()*, *o_r_ic2()*, *or2()*, *or1()*, *r_ic()*, *r()*, *o_r_ic()*, *r_icb()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

Subroutines and functions

subroutine graphout_mod/**graphout**(*time*, *n_r*, *vr*, *vt*, *vp*, *br*, *bt*, *bp*, *sr*, *prer*, *n_theta_start*, *n_theta_block_size*, *lgraphheader*)

Output of components of velocity, magnetic field vector and entropy for graphics.

- n_r: (input) for n_r = 0 a header is written. for n_r > 0 values at radial level n_r are written
- vr...sr: (input) arrays with grid-point values
- n_theta_start : (input) values are written for theta-points : `n_theta_start <= n_theta <= n_theta_start-1+n_theta_block`

Parameters

- **time** *[real,in]*
- **n_r** *[integer,in]* :: radial grod point no.
- **vr** (*nrp*,*) *[real,in]*
- **vt** (*nrp*,*) *[real,in]*
- **vp** (*nrp*,*) *[real,in]*
- **br** (*nrp*,*) *[real,in]*
- **bt** (*nrp*,*) *[real,in]*
- **bp** (*nrp*,*) *[real,in]*
- **sr** (*nrp*,*) *[real,in]*
- **prer** (*nrp*,*) *[real,in]*
- **n_theta_start** *[integer,in]* :: start theta no.
- **n_theta_block_size** *[integer,in]* :: size of theta block
- **lgraphheader** *[logical,inout]*

Called from *fields_average()*

Call to *graph_write()*

subroutine graphout_mod/**graphout_header**(*time*)

Parameters **time** *[real,in]*

subroutine graphout_mod/**graphout_mpi**(*time*, *n_r*, *vr*, *vt*, *vp*, *br*, *bt*, *bp*, *sr*, *prer*, *n_theta_start*, *n_theta_block_size*, *lgraphheader*)

MPI version of the graphOut subroutine (use of MPI_IO)

Parameters

- **time** *[real,in]*
- **n_r** *[integer,in]* :: radial grod point no.
- **vr** (*nrp*,*) *[real,in]*
- **vt** (*nrp*,*) *[real,in]*

- **vp** (*nrp*,*) *[real,in]*
- **br** (*nrp*,*) *[real,in]*
- **bt** (*nrp*,*) *[real,in]*
- **bp** (*nrp*,*) *[real,in]*
- **sr** (*nrp*,*) *[real,in]*
- **prer** (*nrp*,*) *[real,in]*
- **n_theta_start** *[integer,in]* :: start theta no.
- **n_theta_block_size** *[integer,in]* :: size of theta block
- **lgraphheader** *[logical,inout]*

Called from *do_iteration_thetablocking_openmp()*, `do_iteration_thetablocking_seq()`, `do_iteration_thetablocking_shtns()`

Call to *graph_write_mpi()*

subroutine `graphout_mod/`**`graphout_mpi_header`**(*time*, *n_r*, *n_theta_start*, *n_theta_block_size*)

Writes the header (MPI version)

Parameters

- **time** *[real,in]*
- **n_r** *[integer,in]* :: radial grod point no.
- **n_theta_start** *[integer,in]* :: start theta no.
- **n_theta_block_size** *[integer,in]* :: size of theta block

Called from *radialloopg()*

subroutine `graphout_mod/`**`graphout_ic`**(*b_ic*, *db_ic*, *ddb_ic*, *aj_ic*, *dj_ic*, *b*)

Purpose of this subroutine is to write inner core magnetic field onto graphic output file. If the inner core is insulating (l_cond_ic=false) the potential field is calculated from the outer core field at r=r_cmb. This version assumes that the fields are fully local on the rank which is calling this routine (usually rank 0).

Parameters

- **b_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,in]*
- **db_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,in]*
- **ddb_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,in]*
- **aj_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,in]*
- **dj_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,in]*
- **b** (*lm_maxmag*,*n_r_maxmag*) *[complex,in]*

Called from *fields_average()*, *output()*

Call to *legprep_ic()*, *legtf()*, *fft_thetab()*, *graph_write_mpi()*

subroutine `graphout_mod/`**`graph_write`**(*n_phis*, *n_thetas*, *dummy*, *n_graph_file*)

This subroutine writes the data for one theta-band (stored in 'dummy'). Version May, 5, 2000.

Parameters

- **n_phis** *[integer,in]* :: number of logitudes to be printed
- **n_thetas** *[integer,in]* :: number of first colatitude value
- **dummy** (*n_phi_max*,*) *[real,in]* :: data
- **n_graph_file** *[integer,in]* :: output unit

Called from *graphout()*

subroutine graphout_mod/**graph_write_mpi** (*n_phis*, *n_thetas*, *dummy*, *graph_mpi_fh*)

Parameters

- **n_phis** *[integer,in]* :: number of logitudes to be printed
- **n_thetas** *[integer,in]* :: number of first colatitude value
- **dummy** (*n_phi_max*,*) *[real,in]* :: data
- **graph_mpi_fh** *[integer,in]* :: mpi handle of the mpi file

Called from *graphout_mpi()*, *graphout_ic()*

10.16.2 movie.f90

Quick access

Variables *n_frame_work*, *n_md*, *n_movies*, *n_movies_max*, *n_movie_fields_max*, *licfield*, *lstoremov*, *n_movie_const*, *n_movie_fields*, *n_movie_fields_ic*, *n_movie_file*, *n_movie_surface*, *n_movie_type*, *n_movie_field_start*, *n_movie_field_stop*, *n_movie_field_type*, *moviedipcolat*, *moviediplon*, *moviedipstrength*, *moviedipstrengthgeo*, *frames*, *movie_file*, *movie*, *t_movies*, *movie_const*

Routines *get_movie_type()*, *finalize_movie_data()*, *initialize_movie_data()*, *movie_gather_frames_to_rank0()*

Needed modules

- *horizontal_data* (*theta()*, *phi()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *precision_mod*: This module controls the precision used in MagIC
- *useful* (*logwrite()*): library with several useful subroutines
- *output_data* (*n_log_file()*, *log_file()*, *tag()*): This module contains the parameters for output control
- *radial_data* (*n_r_cmb()*, *nrstart()*, *n_r_icb()*, *nrstop()*)
- *parallel_mod*: This module contains the blocking information
- *logic* (*l_movie_oc()*, *l_dtbmovie()*, *l_movie_ic()*, *l_store_frame()*, *l_save_out()*, *l_htmovie()*, *l_movie()*): Module containing the logicals that control the run
- *charmanip* (*length_to_blank()*, *delete_string()*, *capitalize()*, *str2dble()*): This module contains several useful routines to manipule character strings

- *truncation* (*n_r_ic_max()*, *lmoviemem()*, *ldtbmem()*, *minc()*, *n_phi_max()*, *n_r_tot()*, *n_r_max()*, *n_theta_max()*): This module defines the grid points and the truncation
- *constants* (*pi()*, *one()*): module containing constants and parameters used in the code.
- *radial_functions* (*r_cmb()*, *r_icb()*, *r()*, *r_ic()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

- movie_data/**n_frame_work** *[integer,public]*
- movie_data/**n_movies** *[integer,public]*
- movie_data/**n_movie_field_stop** (6,30) *[integer,public]*
- movie_data/**n_movie_surface** (30) *[integer,public]*
- movie_data/**n_movie_field_start** (6,30) *[integer,public]*
- movie_data/**movie_const** (30) *[real,public]*
- movie_data/**n_movies_max** *[integer,parameter*=30/public*]*
  Max no. of different movies
- movie_data/**n_movie_fields_max** *[integer,parameter*=6/public*]*
  Max no. of fields per movie
- movie_data/**n_md** *[integer,public]*
- movie_data/**frames** (:) *[real,allocatable/public]*
- movie_data/**moviediplon** *[real,public]*
- movie_data/**n_movie_fields_ic** (30) *[integer,public]*
- movie_data/**moviedipstrengthgeo** *[real,public]*
- movie_data/**n_movie_field_type** (6,30) *[integer,public]*
- movie_data/**licfield** (30) *[logical,public]*
- movie_data/**movie** (30) *[character,public]*
  Only for input
- movie_data/**n_movie_file** (30) *[integer,public]*
- movie_data/**n_movie_fields** (30) *[integer,public]*
- movie_data/**t_movies** (10000) *[real,public]*
- movie_data/**n_movie_const** (30) *[integer,public]*
- movie_data/**moviedipcolat** *[real,public]*
- movie_data/**movie_file** (30) *[character,public]*
- movie_data/**lstoremov** (30) *[logical,public]*
- movie_data/**moviedipstrength** *[real,public]*
- movie_data/**n_movie_type** (30) *[integer,public]*

Subroutines and functions

subroutine movie_data/**initialize_movie_data**()
This routine is called during the initialization of the code. It allows to:

•Estimate the required memory imprint and allocate the arrays accordingly

•Open the requested movie files

Called from *magic*

Call to *get_movie_type()*, *logwrite()*

subroutine movie_data/**finalize_movie_data**()
Close movie files

Called from *magic*

subroutine movie_data/**get_movie_type**()
Purpose of this subroutine is to identify the different movie types from the input string movies(*). Note that generally blanks are not interpreted and that the interpretation is not case sensitive. In general two informations are needed:

1. A word FIELDINFO that identifies the field to be plotted (e.g. Br for radial magnetic field, see list below)
   Possible keywords are (optional text in brackets):

   •B r[adial] : radial magnetic field

   •B t[heta] : theta component

   •B p[hi] : azimuthal component

   •B h[orizontal] : the two horizontal components

   •B a[ll] : all three components

   •FIELDLINE[S] : field lines of axisymmetric or FL poloidal field for phi=constant

   •AX[ISYMMETRIC] B or AB : axisymmetric phi component of the magnetic field for phi=constant

   •V r[adial] : radial velocity field

   •V t[heta] : theta component

   •V p[hi] : azimuthal component

   •V h[orizontal] : the two horizontal components

   •V a[ll] : all three components

   •STREAMLINE[S] : field lines of axisymmetric or SL : poloidal field for phi=constant

   •AX[ISYMMETRIC] V or AV : axisymmetric phi component of the velocity field for phi=constant

   •V z : z component of velocity at equator and z component of the vorticity at the equator (closest point to equator)

   •Vo z : z-component of vorticity

   •Vo r : r-component of vorticity

   •Vo p : phi-component of vorticity

   •T[emperature] : sic

   •AX[ISYMMETRIC] T or AT : axisymmetric T field for phi=constant

- •Heat t[ransport]: radial derivative of T
- •FL Pro : axisymmetric field line stretching
- •FL Adv : axisymmetric field line advection
- •FL Dif : axisymmetric field line diffusion
- •AB Pro : axisymmetric (tor.) Bphi production
- •AB Dif : axisymmetric (tor.) Bphi diffusion
- •Br Pro : Br production
- •Br Adv : Br advection
- •Br Dif : Br diffusion
- •Jr : Jr production
- •Jr Pro : Jr production + omega effects
- •Jr Adv : Jr advection
- •Jr Dif : Jr diffusion
- •Bz Pol : poloidal Bz
- •Bz Pol Pro : poloidal Bz production
- •Bz Pol Adv : poloidal Bz advection
- •Bz Pol Dif : poloidal Bz diffusion
- •Jz Tor : poloidal Jz
- •Jz Tor Pro : poloidal Jz production
- •Jz Tor Adv : poloidal Jz advection
- •Jz Tor Dif : poloidal Jz diffusion
- •Bp Tor : toriodal Bphi
- •Bp Tor Pro : toriodal Bphi production
- •Bp Tor Adv : toriodal Bphi advection
- •Bp Tor Dif : toriodal Bphi diffusion
- •HEL[ICITY] : sic
- •AX[ISYMMETRIC HELICITY] or AHEL : axisymmetric helicity
- •Bt Tor : toroidal Btheta
- •Pot Tor : toroidal Potential
- •Pol Fieldlines : toroidal Potential
- •Br Shear : azimuthal Shear of Br
- •Lorentz[force] : Lorentz force (only phi component)
- •Br Inv : Inverse field apperance at CMB

2. A second information that identifies the coordinate to be kept constant (surface). E.g. r=number for surface r=constant with number given in units of the total core radius or theta/phi=number with number given in degrees Four keywords are also possible:

   - •CMB : core mantle boundary

- EQ[UATOR] : equatorial plane
- SUR[FACE] : Earth surface (only magnetic field)
- 3[D] : 3D field throughout the OC [and IC for B]

On output the necessary information is coded into integers and is used in this form by further subroutines:

- n_movies = total number of movies
- n_type(n_movie) = movie type:
  - = 1 : Radial magnetic field
  - = 2 : Theta component of magnetic field
  - = 3 : Azimuthal magnetic field
  - = 4 : Horizontal magnetic field
  - = 5 : Total magnetic field (all compnents)
  - = 8 : Axisymmetric azimuthal magnetic field (phi=constant)
  - = 9 : 3d magnetic field
  - = 11 : Radial velocity field
  - = 12 : Theta component of velocity field
  - = 13 : Azimuthal velocity field
  - = 14 : Horizontal velocity field
  - = 15 : Total velocity field (all compnents)
  - = 17 : Scalar field whose contours are the stream lines of the axisymm. poloidal velocity field (phi=constant)
  - = 18 : Axisymmetric azimuthal velocity field (phi=constant)
  - = 19 : 3d velocity field
  - = 20 : z component of vorticity
  - = 21 : Temperature field
  - = 22 : radial conv. heat transport
  - = 23 : helicity
  - = 24 : axisymmetric helicity
  - = 25 : phi component of vorticity
  - = 26 : radial component of vorticity
  - = 28 : axisymmetric Temperature field for phi=const.
  - = 29 : 3d temperature field
  - = 30 : Scalar field whose contours are the fieldlines of the axisymm. poloidal magnetic field (phi=constant)
  - = 31 : field line production
  - = 32 : field line advection
  - = 33 : field line diffusion
  - = 40 : Axisymmetric azimuthal magnetic field (phi=constant)

–= 41 : Axis. Bphi production + omega eff.

–= 42 : Axis. Bphi advection

–= 43 : Axis. Bphi diffusion

–= 44 : Axis. Bphi str.,dyn.,omega,diff.

–= 50 : Bz

–= 51 : Bz production

–= 52 : Bz advection

–= 53 : Bz diffusion

–= 60 : toroidal Bphi

–= 61 : toroidal Bphi production + omega eff.

–= 62 : toroidal Bphi advection

–= 63 : toroidal Bphi diffusion

–= 71 : Br production

–= 72 : Br advection

–= 73 : Br diffusion

–= 80 : Jr

–= 81 : Jr production

–= 82 : Jr advection

–= 83 : Jr diffusion

–= 90 : poloidal Jz pol.

–= 91 : poloidal Jz pol. production

–= 92 : poloidal Jz advection

–= 93 : poloidal Jz diffusion

–= 94 : z component of velovity

–= 95 : toroidal Btheta

–= 96 : toroidal Potential

–= 97 : Function for Poloidal Fieldlines

–= 98 : azimuthal shear of Br

–= 99 : phi component of Lorentz force

–=101 : Stress fields

–=102 : Force fields

–=103 : Br Inverse appearence at CMB

–=110 : radial heat flow

–=111 : Vz and Vorz north/south correlation

–=112 : axisymm dtB tersm for Br and Bp

–=113 : axisymm dSdr

- –=114 : Cylindrically radial magnetic field

•n_movie_surface(n_movie) = defines surface

•n_movie_surface = 1 : r=constant:

- –2 : theta=constant
- –3 : phi=constant
- –-1 : r=constant, Earth surface
- –0 : 3d volume

•n_movie_fields(n_movie) = no. of fields for outer core

•n_movie_fields_ic(n_movie) = no. of fields for inner core

•n_movie_field_type(n_field,n_movie) = defines field

•n_movie_field_type:

- –= 1 : radial magnetic field
- –= 2 : theta comp. of the magnetic field
- –= 3 : azimuthal magnetic field
- –= 4 : radial velocity field
- –= 5 : theta comp. of the velocity field
- –= 6 : azimuthal velocity field
- –= 7 : temperature field
- –= 8 : scalar field for field lines
- –= 9 : axisymm. toroidal mag. field
- –=10 : scalar field for stream lines
- –=11 : axisymm. v_phi
- –=12 : axisymm. T
- –=13 : z-comp. of poloidal Bz
- –=14 : z-comp. of poloidal Jz
- –=15 : z-comp. of velocity
- –=16 : z-comp. of vorticity
- –=17 : radial derivative of T * vr
- –=18 : helicity
- –=19 : axisymmetric helicity
- –=20 : axisymm field-line production
- –=21 : axisymm field-line advection
- –=22 : axisymm field-line diffusion
- –=23 : axisymm Bphi production
- –=24 : axisymm Bphi omega effect
- –=25 : axisymm Bphi advection

–=26 : axisymm Bphi diffusion

–=27 : Br production

–=28 : Br advection

–=29 : Br diffusion

–=30 : Jr

–=31 : Jr production

–=32 : Jr omega effect

–=33 : Jr advection

–=34 : Jr diffusion

–=35 : poloidal Bz production

–=36 : poloidal Bz advection

–=37 : poloidal Bz diffusion

–=38 : poloidal Jz production

–=39 : poloidal Jz omega effect

–=40 : poloidal Jz advection

–=41 : poloidal Jz diffusion

–=42 : toroidal Bp

–=43 : toroidal Bp production

–=44 : toroidal Bp omega effect

–=45 : toroidal Bp advection

–=46 : toroidal Bp diffusion

–=47 : phi-comp. of vorticity

–=48 : r-comp. of vorticity

–=49 : toroidal Bp omega effect

–=50 : toroidal Bt

–=51 : toroidal Potential

–=52 : poloidal Fieldlines in theta=const

–=53 : Br dr (vp/(r sin(theta))

–=54 : phi Lorentz force

–=61 : AS phi reynolds stress force

–=62 : AS phi advective stress force

–=63 : AS phi viscous stress force

–=64 : AS phi Lorentz force

–=66 : time derivative of axisym. v phi

–=67 : relative strength of axisym. v phi

–=81 : Br inverse appearence at CMB

–=91 : radial derivative of T

–=92 : Vz north/south correlation

–=93 : Vorz north/south correlation

–=94 : Hel north/south correlation

–=101: AS poloidal Br production

–=102: AS poloidal Br dynamo term

–=103: AS poloidal Br diffusion

–=104: AS toroidal Bp production

–=105: AS toroidal Bp dynamo term

–=106: AS toroidal Bp omega effect

–=107: AS toroidal Bp diffusion

–=108: Bs

•n_movie_field_start(n_field,n_movie) = defines where first element of a field is stored in `frames(*)`

•n_movie_field_stop(n_field,n_movie) = defines where last element of a field is stored in `frames(*)`

•The subroutine also defines appropriate file names for the movie files. These generally have the form TYPE_mov.TAG

Called from *initialize_movie_data()*

Call to *delete_string()*, *capitalize()*, *length_to_blank()*, *str2dble()*

subroutine `movie_data/`**movie_gather_frames_to_rank0**`()`
MPI communicators for movie files

Called from *output()*

10.16.3 out_movie_file.f90

Quick access

Routines *get_b_surface()*, *get_fl()*, *write_movie_frame()*, *store_movie_frame()*, *store_fields_sur()*, *get_sl()*, *store_fields_p()*, *store_fields_3d()*, *store_fields_t()*, *store_fields_r()*

Needed modules

- *horizontal_data* (*dplm()*, *phi()*, *dlh()*, *d_l()*, *theta_ord()*, *osn1()*, *o_sin_theta_e2()*, *o_sin_theta()*, *dphi()*, *plm()*, *n_theta_cal2ord()*, *sintheta()*, *costheta()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *movie_data* (*movie_const()*, *n_movie_fields_ic()*, *n_movies()*, *n_movie_field_type()*, *n_movie_surface()*, *n_movie_file()*, *n_movie_field_stop()*, *n_movie_field_start()*, *n_movie_fields()*, *t_movies()*, *moviedipstrengthgeo()*, *n_movie_const()*, *moviedipcolat()*, *movie_file()*, *frames()*, *lstoremov()*, *moviedipstrength()*, *moviediplon()*, *n_movie_type()*)

- *output_data* (*runid()*): This module contains the parameters for output control
- *fields* (*w_rloc()*, *b_ic()*, *b_rloc()*, *b()*): This module contains the potential fields and their radial derivatives
- *fft* (*fft_thetab()*): This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5
- *out_dtb_frame* (*write_dtb_frame()*)
- *radial_data* (*n_r_icb()*)
- *precision_mod*: This module controls the precision used in MagIC
- *constants* (*zero()*, *two()*, *one()*): module containing constants and parameters used in the code.
- *num_param* (*tscale()*, *vscale()*): Module containing numerical and control parameters
- *physical_parameters* (*pr()*, *prmag()*, *lffac()*, *ek()*, *radratio()*, *ra()*): Module containing the physical parameters
- *truncation* (*n_r_ic_max()*, *l_max()*, *n_r_ic_maxmag()*, *minc()*, *n_phi_max()*, *lm_maxmag()*, *n_r_max()*, *n_m_max()*, *lm_max()*, *nrp()*, *n_r_maxmag()*, *n_theta_max()*): This module defines the grid points and the truncation
- *blocking* (*lm2()*, *lm2l()*, *nfs()*): Module containing blocking information
- *logic* (*l_save_out()*, *l_cond_ic()*): Module containing the logicals that control the run
- *radial_functions* (*orho1()*, *orho2()*, *r_cmb()*, *or4()*, *or2()*, *or3()*, *or1()*, *r_ic()*, *beta()*, *r()*, *r_surface()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

Subroutines and functions

subroutine out_movie/**store_movie_frame**(*n_r*, *vr*, *vt*, *vp*, *br*, *bt*, *bp*, *sr*, *drsr*, *dvrdp*, *dvpdr*, *dvtdr*, *dvrdt*, *cvr*, *cbr*, *cbt*, *n_theta_start*, *n_theta_block*, *bcmb*)

Controls output of movie frames. Usually called from radialLoop.

Parameters

- **n_r** *[integer,in]* :: radial grid point no.
- **vr** (*nrp*,*) *[real,in]*
- **vt** (*nrp*,*) *[real,in]*
- **vp** (*nrp*,*) *[real,in]*
- **br** (*nrp*,*) *[real,in]*
- **bt** (*nrp*,*) *[real,in]*
- **bp** (*nrp*,*) *[real,in]*
- **sr** (*nrp*,*) *[real,in]*
- **drsr** (*nrp*,*) *[real,in]*
- **dvrdp** (*nrp*,*) *[real,in]*
- **dvpdr** (*nrp*,*) *[real,in]*

- **dvtdr** (*nrp*,*) *[real,in]*
- **dvrdt** (*nrp*,*) *[real,in]*
- **cvr** (*nrp*,*) *[real,in]*
- **cbr** (*nrp*,*) *[real,in]*
- **cbt** (*nrp*,*) *[real,in]*
- **n_theta_start** *[integer,in]* :: start theta no.
- **n_theta_block** *[integer,in]* :: size of theta block
- **bcmb** (*lm_max*) *[complex,in]*

Called from *do_iteration_thetablocking_openmp()*, do_iteration_thetablocking_seq(), do_iteration_thetablocking_shtns()

Call to *store_fields_sur()*, *store_fields_3d()*, *store_fields_r()*, *store_fields_t()*, *store_fields_p()*

subroutine out_movie/**write_movie_frame**(*n_frame*, *time*, *b*, *db*, *aj*, *dj*, *b_ic*, *db_ic*, *aj_ic*, *dj_ic*, *omega_ic*, *omega_ma*)

Writes different movie frames into respective output files. Called from rank 0 with full arrays in standard LM order.

Parameters

- **n_frame** *[integer,in]*
- **time** *[real,in]*
- **b** (*lm_maxmag*,*n_r_maxmag*) *[complex,in]*
- **db** (*lm_maxmag*,*n_r_maxmag*) *[complex,in]*
- **aj** (*lm_maxmag*,*n_r_maxmag*) *[complex,in]*
- **dj** (*lm_maxmag*,*n_r_maxmag*) *[complex,in]*
- **b_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,in]*
- **db_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,in]*
- **aj_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,in]*
- **dj_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,in]*
- **omega_ic** *[real,in]*
- **omega_ma** *[real,in]*

Called from *output()*

Call to *write_dtb_frame()*

subroutine out_movie/**store_fields_sur**(*n_store_last*, *n_field_type*, *n_theta_start*, *n_theta_block*, *bcmb*)

Purpose of this subroutine is to store movie frames for surfaces r=const. into array frame(,)

Parameters

- **n_store_last** *[integer,in]* :: Start position for storing -1
- **n_field_type** *[integer,in]* :: Defines field type

- **n_theta_start** *[integer,in]* :: Beginning of theta block
- **n_theta_block** *[integer,in]* :: Size of theta block
- **bcmb** (*lm_max*) *[complex,in]*

Called from *store_movie_frame()*

Call to *get_b_surface()*

subroutine out_movie/**store_fields_r**(*vr*, *vt*, *vp*, *br*, *bt*, *bp*, *sr*, *drsr*, *dvrdp*, *dvpdr*, *dvtdr*, *dvrdt*, *cvr*, *n_r*, *n_store_last*, *n_field_type*, *n_theta_start*, *n_theta_block*)

Purpose of this subroutine is to store movie frames for surfaces r=const. into array frame(,)

Parameters

- **vr** (*nrp*,*) *[real,in]*
- **vt** (*nrp*,*) *[real,in]*
- **vp** (*nrp*,*) *[real,in]*
- **br** (*nrp*,*) *[real,in]*
- **bt** (*nrp*,*) *[real,in]*
- **bp** (*nrp*,*) *[real,in]*
- **sr** (*nrp*,*) *[real,in]*
- **drsr** (*nrp*,*) *[real,in]*
- **dvrdp** (*nrp*,*) *[real,in]*
- **dvpdr** (*nrp*,*) *[real,in]*
- **dvtdr** (*nrp*,*) *[real,in]*
- **dvrdt** (*nrp*,*) *[real,in]*
- **cvr** (*nrp*,*) *[real,in]*
- **n_r** *[integer,in]*
- **n_store_last** *[integer,in]* :: Start position in frame(*)-1
- **n_field_type** *[integer,in]* :: Defines field type
- **n_theta_start** *[integer,in]* :: Beginning of theta block
- **n_theta_block** *[integer,in]* :: Size of theta block

Called from *store_movie_frame()*

subroutine out_movie/**store_fields_p**(*vr*, *vt*, *vp*, *br*, *bp*, *bt*, *sr*, *drsr*, *dvrdp*, *dvpdr*, *dvtdr*, *dvrdt*, *cvr*, *cbr*, *cbt*, *n_r*, *n_store_last*, *n_field_type*, *n_phi_const*, *n_field_size*, *n_theta_start*, *n_theta_block*)

Purpose of this subroutine is to store movie frames for surfaces phi=const. into array frames(,)

Parameters

- **vr** (*nrp*,*) *[real,in]*
- **vt** (*nrp*,*) *[real,in]*
- **vp** (*nrp*,*) *[real,in]*

- **br** (*nrp*,*) *[real,in]*
- **bp** (*nrp*,*) *[real,in]*
- **bt** (*nrp*,*) *[real,in]*
- **sr** (*nrp*,*) *[real,in]*
- **drsr** (*nrp*,*) *[real,in]*
- **dvrdp** (*nrp*,*) *[real,in]*
- **dvpdr** (*nrp*,*) *[real,in]*
- **dvtdr** (*nrp*,*) *[real,in]*
- **dvrdt** (*nrp*,*) *[real,in]*
- **cvr** (*nrp*,*) *[real,in]*
- **cbr** (*nrp*,*) *[real,in]*
- **cbt** (*nrp*,*) *[real,in]*
- **n_r** *[integer,in]* :: No. of radial point
- **n_store_last** *[integer,in]* :: Start position in frame(*)-1
- **n_field_type** *[integer,in]* :: Defines field type
- **n_phi_const** *[integer,in]* :: No. of surface phi
- **n_field_size** *[integer,in]* :: Size of field
- **n_theta_start** *[integer,in]* :: Beginning of theta block
- **n_theta_block** *[integer,in]* :: Size of theta block

Called from *store_movie_frame()*

Call to *get_fl()*, *get_sl()*

subroutine out_movie/**store_fields_t** (*vr*, *vt*, *vp*, *br*, *bt*, *bp*, *sr*, *drsr*, *dvrdp*, *dvpdr*, *dvtdr*, *dvrdt*, *cvr*, *cbt*, *n_r*, *n_store_last*, *n_field_type*, *n_theta_const*, *n_theta*)

Purpose of this subroutine is to store movie frames for surfaces r=const. into array frame(,)

Parameters

- **vr** (*nrp*,*) *[real,in]*
- **vt** (*nrp*,*) *[real,in]*
- **vp** (*nrp*,*) *[real,in]*
- **br** (*nrp*,*) *[real,in]*
- **bt** (*nrp*,*) *[real,in]*
- **bp** (*nrp*,*) *[real,in]*
- **sr** (*nrp*,*) *[real,in]*
- **drsr** (*nrp*,*) *[real,in]*
- **dvrdp** (*nrp*,*) *[real,in]*
- **dvpdr** (*nrp*,*) *[real,in]*
- **dvtdr** (*nrp*,*) *[real,in]*

- **dvrdt** (*nrp*,*) *[real,in]*
- **cvr** (*nrp*,*) *[real,in]*
- **cbt** (*nrp*,*) *[real,in]*
- **n_r** *[integer,in]* :: No. of radial grid point
- **n_store_last** *[integer,in]* :: Position in frame(*)-1
- **n_field_type** *[integer,in]* :: Defines field
- **n_theta_const** *[integer,in]* :: No. of theta to be stored
- **n_theta** *[integer,in]* :: No. of theta in block

Called from *store_movie_frame()*

subroutine out_movie/**store_fields_3d**(*vr*, *vt*, *vp*, *br*, *bt*, *bp*, *sr*, *drsr*, *dvrdp*, *dvpdr*, *dvtdr*, *dvrdt*, *cvr*, *cbr*, *cbt*, *n_r*, *n_store_last*, *n_field_type*, *n_theta_start*, *n_theta_block*)

Purpose of this subroutine is to store movie frames for surfaces r=const. into array frame(,)

Parameters

- **vr** (*nrp*,*) *[real,in]*
- **vt** (*nrp*,*) *[real,in]*
- **vp** (*nrp*,*) *[real,in]*
- **br** (*nrp*,*) *[real,in]*
- **bt** (*nrp*,*) *[real,in]*
- **bp** (*nrp*,*) *[real,in]*
- **sr** (*nrp*,*) *[real,in]*
- **drsr** (*nrp*,*) *[real,in]*
- **dvrdp** (*nrp*,*) *[real,in]*
- **dvpdr** (*nrp*,*) *[real,in]*
- **dvtdr** (*nrp*,*) *[real,in]*
- **dvrdt** (*nrp*,*) *[real,in]*
- **cvr** (*nrp*,*) *[real,in]*
- **cbr** (*nrp*,*) *[real,in]*
- **cbt** (*nrp*,*) *[real,in]*
- **n_r** *[integer,in]* :: No. of radial grid point
- **n_store_last** *[integer,in]* :: Position in frame(*)-1
- **n_field_type** *[integer,in]* :: Defines field
- **n_theta_start** *[integer,in]* :: No. of first theta to block
- **n_theta_block** *[integer,in]* :: Size of theta block

Called from *store_movie_frame()*

subroutine out_movie/**get_sl**(*sl*, *n_r*, *n_theta_start*, *n_theta_block*)

Return field sl whose contourlines are the stream lines of the axisymmetric poloidal velocity field. sl(r,theta)=d_theta v(r,theta,m=0)/r

Parameters

- **sl** (*) *[real,out]* :: Field for field lines
- **n_r** *[integer,in]* :: No. of radial grid point
- **n_theta_start** *[integer,in]* :: No. of theta to start with
- **n_theta_block** *[integer,in]* :: Size of theta block

Called from *store_fields_p()*

subroutine out_movie/**get_fl** (*fl*, *n_r*, *n_theta_start*, *n_theta_block*, *l_ic*)

Return field fl whose contourlines are the fields lines of the axisymmetric poloidal mangetic field.

fl(r,theta)=d_theta b(r,theta,m=0)/r

Parameters

- **fl** (*) *[real,out]* :: Field for field lines
- **n_r** *[integer,in]* :: No. of radial grid point
- **n_theta_start** *[integer,in]* :: No. of theta to start with
- **n_theta_block** *[integer,in]* :: Size of theta block
- **l_ic** *[logical,in]* :: =true if inner core field

Called from *store_fields_p()*, *store_movie_frame_ic()*

subroutine out_movie/**get_b_surface** (*b_r*, *b_t*, *b_p*, *bcmb*, *n_theta_start*, *n_theta_block*)

Upward continuation of laplacian field to Earths surface. Field is given by poloidal harmonic coefficients b at CMB. Spherical harmonic transforms of upward continued field to r/theta/phi vector components for all logitudes and latitude are returned in br/bt/bp. Note that this routine given the real components of the magnetic fields while other transforms in the code provide only:

r**2 br, r**2 sin(theta) bt, r**2 sin(theta) bp

Parameters

- **b_r** (*nrp*,*) *[real,out]* :: Radial magnetic field in (phi,theta)-space
- **b_t** (*nrp*,*) *[real,out]* :: Latitudinal magnetic field
- **b_p** (*nrp*,*) *[real,out]* :: Azimuthal magnetic field.
- **bcmb** (*lm_max*) *[complex,in]*
- **n_theta_start** *[integer,in]* :: No. of theta to start with
- **n_theta_block** *[integer,in]* :: Size of theta block

Called from *store_fields_sur()*

Call to *fft_thetab()*

10.16.4 store_movie_IC.f90

Quick access

Routines *store_movie_frame_ic()*

Needed modules

- *horizontal_data* (*dplm()*, *plm()*, *dlh()*, *n_theta_cal2ord()*, *o_sin_theta()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *movie_data* (*n_movie_fields_ic()*, *n_movies()*, *n_movie_field_stop()*, *n_movie_surface()*, *n_movie_field_type()*, *n_movie_field_start()*, *n_movie_fields()*, *n_movie_const()*, *frames()*, *n_movie_type()*)
- *out_movie* (*get_fl()*)
- *fft* (*fft_thetab()*): This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5
- *legendre_spec_to_grid* (*legtf()*)
- *radial_data* (*n_r_icb()*)
- *precision_mod*: This module controls the precision used in MagIC
- *logic* (*l_cond_ic()*): Module containing the logicals that control the run
- *physical_parameters* (*lffac()*): Module containing the physical parameters
- *truncation* (*n_r_ic_max()*, *lm_maxmag()*, *n_r_ic_maxmag()*, *minc()*, *n_phi_max()*, *l_max()*, *lm_max()*, *nrp()*, *n_r_maxmag()*, *n_theta_max()*): This module defines the grid points and the truncation
- *leg_helper_mod* (*legprep_ic()*)
- *blocking* (*nthetabs()*, *nfs()*, *sizethetab()*): Module containing blocking information
- *constants* (*one()*): module containing constants and parameters used in the code.
- *radial_functions* (*r_icb()*, *r_ic()*, *o_r_ic2()*, *o_r_ic()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

Subroutines and functions

subroutine out_movie_ic/**store_movie_frame_ic**(*b*, *b_ic*, *db_ic*, *ddb_ic*, *aj_ic*, *dj_ic*)

Controls storage of IC magnetic field in movie frame.

Parameters

- **b** (*lm_maxmag*,*n_r_maxmag*) *[complex,in]*
- **b_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,in]*
- **db_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,in]*
- **ddb_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,in]*

- **aj_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,in]*
- **dj_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,in]*

Called from *output()*

Call to *legprep_ic()*, *legtf()*, *fft_thetab()*, *get_fl()*

10.16.5 out_coeff.f90

Description

This module contains the subroutines that calculate the Bcmb files and the [B|V|T]_coeff_r files

Quick access

Routines *write_bcmb()*, *write_coeff_r()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *logic* (*l_save_out()*): Module containing the logicals that control the run

Variables

Subroutines and functions

subroutine out_coeff/**write_bcmb**(*time*, *b*, *llm*, *ulm*, *l_max*, *l_max_cmb*, *minc*, *lm2*, *n_cmb_sets*, *cmb_file*, *n_cmb_file*)

Each call of this subroutine writes time and the poloidal magnetic potential coefficients b at the CMB up to degree and order l_max_cmb at the end of output file $cmb_file. The parameters l_max_cmb, minc and the number of stored coeffs. are written into the first line of $cmb_file. Each further set contains:

```
time,
real(b(l=0,m=0)),imag(b(l=0,m=0)),
real(b(l=1,m=0)),imag(b(l=1,m=0)),
```

Real and imaginary part of b(*) for all orders m<=l are written for a specific degree l, then for the degrees l+1, l+2, l_max_cmb.

Parameters

- **time** *[real,in]* :: Time
- **b** (*ulm*-*llm*+1) *[complex,in]*
- **llm** *[integer,in]*
- **ulm** *[integer,in]* :: Poloidal field potential
- **l_max** *[integer,in,]* :: Gives position of (l,m) coeff
- **l_max_cmb** *[integer,inout]* :: Max degree of output

- **minc** *[integer,in]* :: Basic wave-number
- **lm2** (*l_max* + 1,*l_max* + 1) *[integer,in]*
- **n_cmb_sets** *[integer,inout]* :: Total no. of cmb sets,
- **cmb_file** *[character,in]* :: Name of output file
- **n_cmb_file** *[integer,in]* :: Output unit for $cmb_file

Called from *fields_average()*, *output()*

subroutine out_coeff/**write_coeff_r**(*time*, *w*, *dw*, *ddw*, *z*, *r*, *llm*, *ulm*, *l_max*, *l_max_r*, *minc*, *lm2*, *n_sets*, *file*, *n_file*, *nvbs*)

Each call of this subroutine writes time and the poloidal and toroidal coeffitients w,dw,z at a specific radius up to degree and order l_max_r at the end of output file $file. If the input is magnetic field (nVBS=2) ddw is stored as well. If the input is entropy field (nVBS=3) only ddw=s is stored. The parameters l_max_r, minc, the number of stored coeffs and radius in the outer core are written into the first line of $file. Each further set contains:

```
time,
real(w(l=0,m=0)),imag(w(l=0,m=0)),
real(w(l=1,m=0)),imag(w(l=1,m=0)),
...
real(dw(l=0,m=0)),imag(dw(l=0,m=0)),
real(dw(l=1,m=0)),imag(dw(l=1,m=0)),
...
real(z(l=0,m=0)),imag(z(l=0,m=0)),
real(z(l=1,m=0)),imag(z(l=1,m=0)),
...
real(ddw(l=0,m=0)),imag(ddw(l=0,m=0)),
real(ddw(l=1,m=0)),imag(ddw(l=1,m=0)),
```

Real and imaginary part of w(*) for all orders m<=l are written for a specific degree l, then for the degrees l+1, l+2, l_max_r.

Parameters

- **time** *[real,in]* :: Time
- **w** (*ulm*-*llm*+1) *[complex,in]*
- **dw** (*ulm*-*llm*+1) *[complex,in]*
- **ddw** (*ulm*-*llm*+1) *[complex,in]*
- **z** (*ulm*-*llm*+1) *[complex,in]*
- **r** *[real,in]* :: radius of coeffs
- **llm** *[integer,in]*
- **ulm** *[integer,in]* :: Toroidal field potential
- **l_max** *[integer,in,]* :: Max degree of b(,)
- **l_max_r** *[integer,inout]* :: Max degree of output
- **minc** *[integer,in]* :: Basic wave-number
- **lm2** (*l_max* + 1,*l_max* + 1) *[integer,in]*
- **n_sets** *[integer,inout]* :: Total no. of cmb sets,

- **file** *[character,in]* :: Name of output file
- **n_file** *[integer,in]* :: Output unit for $file
- **nvbs** *[integer,in]* :: True if output is flow

Called from *output()*

10.16.6 store_pot.f90

Description

This module contains the subroutines that can be used to write unformatted fortran files that contain the flow/magnetic field potentials (in both Chebyshev and spectral space)

Quick access

Routines *storepot()*, *storepotw()*

Needed modules

- *parallel_mod* (rank()): This module contains the blocking information
- *cosine_transform_odd*
- *output_data* (*tag()*): This module contains the parameters for output control
- *lmloop_data* (*llm()*, *ulm()*)
- *precision_mod*: This module controls the precision used in MagIC
- *communications* (*gather_from_lo_to_rank0()*)
- *logic* (*l_cond_ic()*): Module containing the logicals that control the run
- *physical_parameters* (*pr()*, *prmag()*, *ek()*, *sigma_ratio()*, *radratio()*, *ra()*): Module containing the physical parameters
- *truncation* (*n_r_ic_max()*, *n_cheb_ic_max()*, *minc()*, *l_max()*, *n_r_max()*, *lm_max()*, *n_cheb_max()*): This module defines the grid points and the truncation
- *constants* (*two()*, *half()*): module containing constants and parameters used in the code.
- *radial_functions* (*chebt_oc()*, *chebt_ic()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

Subroutines and functions

subroutine store_pot_mod/**storepot** (*time*, *b*, *aj*, *b_ic*, *aj_ic*, *npotsets*, *root*, *omega_ma*, *omega_ic*)

Parameters

- **time** *[real,in]*
- **b** (*lm_max*,*n_r_max*) *[complex,in]*

- **aj** (*lm_max*,*n_r_max*) *[complex,in]*
- **b_ic** (*lm_max*,*n_r_ic_max*) *[complex,in]*
- **aj_ic** (*lm_max*,*n_r_ic_max*) *[complex,in]*
- **npotsets** *[integer,inout]*
- **root** *[character,in]*
- **omega_ma** *[real,in]*
- **omega_ic** *[real,in]*

Called from `output()`

subroutine store_pot_mod/**storepotw**(*time*, *b*, *aj*, *b_ic*, *aj_ic*, *workа*, *workb*, *workc*, *npotsets*, *root*, *omega_ma*, *omega_ic*)

Parameters

- **time** *[real,in]*
- **b** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **aj** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **b_ic** (*ulm*-(*llm*)+1,*n_r_ic_max*) *[complex,in]*
- **aj_ic** (*ulm*-(*llm*)+1,*n_r_ic_max*) *[complex,in]*
- **workа** (*ulm*-(*llm*)+1,*n_r_max*) *[complex]*
- **workb** (*ulm*-(*llm*)+1,*n_r_max*) *[complex]*
- **workc** (*ulm*-(*llm*)+1,*n_r_max*) *[complex]*
- **npotsets** *[integer,inout]*
- **root** *[character,in]*
- **omega_ma** *[real,in]*
- **omega_ic** *[real,in]*

Called from `fields_average()`

Call to `gather_from_lo_to_rank0()`

10.16.7 field_average.f90

Description

This module is used when one wants to store time-averaged quantities

Quick access

Variables `aj_ave_global`, `db_ave_global`, `dw_ave_global`, `p_ave_global`, `s_ave_global`, `w_ave_global`, `z_ave_global`, `aj_ave`, `aj_ic_ave`, `b_ave`, `b_ic_ave`, `p_ave`, `s_ave`, `w_ave`, `z_ave`

Routines `initialize_fields_average_mod()`, `fields_average()`

Needed modules

- *horizontal_data* (*dplm()*, *plm()*, *dlh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *fft* (*fft_thetab()*): This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5
- *lmloop_data* (*ulmmag()*, *llmmag()*, *llm()*, *ulm()*)
- *radial_der_even* (*get_drns_even()*, *get_ddrns_even()*)
- *radial_der* (*get_drns()*): Radial derivatives functions
- *graphout_mod* (*graphout()*, *graphout_ic()*)
- *magnetic_energy* (*get_e_mag()*)
- *store_pot_mod* (*storepotw()*): This module contains the subroutines that can be used to write unformatted fortran files that contain the flow/magnetic field potentials (in both Chebyshev and spectral space)
- *parallel_mod* (`rank()`): This module contains the blocking information
- *radial_functions* (*drx()*, *dr_fac_ic()*, *chebt_ic_even()*, *r()*, *chebt_oc()*, *chebt_ic()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *leg_helper_mod* (*legprep()*)
- *precision_mod*: This module controls the precision used in MagIC
- *kinetic_energy* (*get_e_kin()*)
- *out_coeff* (*write_bcmb()*): This module contains the subroutines that calculate the Bcmb files and the [B|V|T]_coeff_r files
- *blocking* (*lmstartb()*, *lm2()*, *nthetabs()*, *sizethetab()*, *nfs()*, *lmstopb()*): Module containing blocking information
- *constants* (*zero()*, *vol_ic()*, *vol_oc()*, *one()*): module containing constants and parameters used in the code.
- *truncation*: This module defines the grid points and the truncation
- *output_data* (*l_max_cmb()*, *n_graphs()*, *graph_file()*, *n_graph_file()*, *tag()*, *nlf()*, *log_file()*): This module contains the parameters for output control
- *spectra* (*spectrum()*, *spectrum_temp()*)
- *legendre_spec_to_grid* (*legtf()*)
- *radial_data* (*n_r_cmb()*)
- *communications* (*get_global_sum()*, *gather_from_lo_to_rank0()*, *gather_all_from_lo_to_rank0()*, *gt_ic()*, *gt_oc()*)
- *logic* (*l_heat()*, *l_save_out()*, *l_cond_ic()*, *l_mag()*, *l_conv()*): Module containing the logicals that control the run

Variables

- `fields_average_mod/`**`z_ave`** (:,:) *[complex,private/allocatable]*
- `fields_average_mod/`**`b_ic_ave`** (:,:) *[complex,private/allocatable]*

- fields_average_mod/**db_ave_global** (:) *[complex,private/allocatable]*
- fields_average_mod/**s_ave_global** (:) *[complex,private/allocatable]*
- fields_average_mod/**z_ave_global** (:) *[complex,private/allocatable]*
- fields_average_mod/**p_ave** (:,:) *[complex,private/allocatable]*
- fields_average_mod/**b_ave** (:,:) *[complex,private/allocatable]*
- fields_average_mod/**w_ave_global** (:) *[complex,private/allocatable]*
- fields_average_mod/**p_ave_global** (:) *[complex,private/allocatable]*
- fields_average_mod/**aj_ave_global** (:) *[complex,private/allocatable]*
- fields_average_mod/**s_ave** (:,:) *[complex,private/allocatable]*
- fields_average_mod/**aj_ave** (:,:) *[complex,private/allocatable]*
- fields_average_mod/**w_ave** (:,:) *[complex,private/allocatable]*
- fields_average_mod/**aj_ic_ave** (:,:) *[complex,private/allocatable]*
- fields_average_mod/**dw_ave_global** (:) *[complex,private/allocatable]*

Subroutines and functions

subroutine fields_average_mod/**initialize_fields_average_mod**()

Called from *magic*

subroutine fields_average_mod/**fields_average**(*nave*, *l_stop_time*, *time_passed*, *time_norm*, *omega_ic*, *omega_ma*, *w*, *z*, *p*, *s*, *b*, *aj*, *b_ic*, *aj_ic*)

This subroutine averages fields b and v over time.

Parameters

- **nave** *[integer,in]* :: number for averaged time steps
- **l_stop_time** *[logical,in]* :: true if this is the last time step
- **time_passed** *[real,in]* :: time passed since last log
- **time_norm** *[real,in]* :: time passed since start of time loop
- **omega_ic** *[real,in]*
- **omega_ma** *[real,in]*
- **w** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **z** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **p** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **s** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **b** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,in]*
- **aj** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,in]*
- **b_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,in]*
- **aj_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,in]*

Called from *output()*

Call to *get_drns()*, *get_ddrns_even()*, *get_drns_even()*, *spectrum()*, *spectrum_temp()*, *get_e_kin()*, *get_e_mag()*, *graphout()*, *gather_all_from_lo_to_rank0()*, *gather_from_lo_to_rank0()*, *legprep()*, *legtf()*, *fft_thetab()*, *graphout_ic()*, *write_bcmb()*, *storepotw()*

10.17 IO: RMS force balance, torsional oscillations, misc

10.17.1 RMS.f90

Description

This module contains the calculation of thr RMS force balance and induction terms.

Quick access

Variables *n_cheb_maxc*, *ncut*, *n_r_maxc*, *chebt_rms*, *eps*, *advrmsl_sd*, *advrmsl_ta*, *advrmssd*, *arcrmsl_sd*, *arcrmsl_ta*, *arcrmssd*, *buormsl_sd*, *buormsl_ta*, *buormssd*, *ciarmsl_sd*, *ciarmsl_ta*, *ciarmssd*, *clfrmsl_sd*, *clfrmsl_ta*, *clfrmssd*, *corrmsl_sd*, *corrmsl_ta*, *corrmssd*, *difrmsl_sd*, *difrmsl_ta*, *difrmssd*, *dtvrmsl_sd*, *dtvrmsl_ta*, *dtvrmssd*, *geormsl_sd*, *geormsl_ta*, *geormssd*, *lfrmsl_sd*, *lfrmsl_ta*, *lfrmssd*, *magrmsl_sd*, *magrmsl_ta*, *magrmssd*, *plfrmsl_sd*, *plfrmsl_ta*, *plfrmssd*, *prermsl_sd*, *prermsl_ta*, *prermssd*, *rc*, *dr_facc*, *adv2hint*, *arc2hint*, *buo2hint*, *cia2hint*, *clf2hint*, *cor2hint*, *difpollmr*, *dtbpollmr*, *dtvpollmr*, *geo2hint*, *lf2hint*, *mag2hint*, *plf2hint*, *pre2hint*, *difpol2hint*, *diftor2hint*, *dtbpol2hint*, *dtbtor2hint*, *dtvpol2hint*, *dtvtor2hint*

Routines *initialize_rms()*, *zerorms()*, *init_rnb()*, *dtbrms()*, *dtvrms()*

Needed modules

- *horizontal_data* (*phi()*, *theta_ord()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *precision_mod*: This module controls the precision used in MagIC
- *cosine_transform_odd*
- *output_data* (*rcut()*, *n_dtbrms_file()*, *tag()*, *runid()*, *dtvrms_file()*, *dtbrms_file()*, *n_dtvrms_file()*, *rdea()*): This module contains the parameters for output control
- *useful* (*getmsd2()*): library with several useful subroutines
- *radial_der* (*get_drns()*, *get_dr()*): Radial derivatives functions
- *rms_helpers* (*hint2dpollm()*, *get_poltorrms()*, *hint2dpol()*, *get_ras()*, *get_paslm()*): This module contains several useful subroutines required to compute RMS diagnostics
- *num_param* (*tscale()*): Module containing numerical and control parameters

- *dtb_mod* (*tdifasrms()*, *pstrlm()*, *tadvrms()*, *pdiflm()*, *tstrrms()*, *tadvlm()*, *pdifasrms()*, *tstrasrms()*, *tdiflm()*, *tomeasrms()*, *tadvasrms()*, *padvrms()*, *pstrasrms()*, *padvlm()*, *tstrlm()*, *padvasrms()*, *pdifrms()*, *pstrrms()*, *tomerms()*, *tdifrms()*, *tomelm()*): This module contains magnetic field stretching and advection terms plus a separate omega-effect. It is used for movie output....
- *integration* (*rint()*, *rint_r()*): Radial integration functions
- *radial_data* (*nrstart()*, *nrstop()*)
- *parallel_mod*: This module contains the blocking information
- *communications* (*myallgather()*)
- *logic* (*l_corr()*, *l_heat()*, *l_mag_lf()*, *l_save_out()*, *l_conv_nl()*, *l_conv()*): Module containing the logicals that control the run
- *chebyshev_polynoms_mod* (*cheb_grid()*)
- *physical_parameters* (*pr()*, *prmag()*, *radratio()*, *ra()*, *ek()*): Module containing the physical parameters
- *radial_functions* (*r()*, *ndd_costf1()*, *ndi_costf1()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *blocking* (*lm2()*, *nthetabs()*, *sizethetab()*, *lm2m()*, *nfs()*, *lo_map()*, *st_map()*): Module containing blocking information
- *constants* (*third()*, *one()*, *four()*, *zero()*, *half()*, *vol_oc()*, *pi()*): module containing constants and parameters used in the code.
- *truncation* (*l_max()*, *minc()*, *n_phi_max()*, *lm_maxmag()*, *n_r_max_dtb()*, *n_r_max()*, *lm_max_dtb()*, *lm_max()*, *n_cheb_max()*, *n_r_maxmag()*, *n_theta_max()*): This module defines the grid points and the truncation

Variables

- rms/**buormsl_sd** (:) *[real,private/allocatable]*
- rms/**dtbpol2hint** (:,:,:) *[real,allocatable/public]*
- rms/**geo2hint** (:,:) *[real,allocatable/public]*
- rms/**dtvrmssd** (:) *[real,private/allocatable]*
- rms/**corrmsl_ta** (:) *[real,private/allocatable]*
- rms/**dtvtor2hint** (:,:,:) *[real,allocatable/public]*
- rms/**advrmsl_ta** (:) *[real,private/allocatable]*
- rms/**cia2hint** (:,:) *[real,allocatable/public]*
- rms/**plfrmsl_sd** (:) *[real,private/allocatable]*
- rms/**ncut** *[integer,public]*
  Number of points for the cut-off
- rms/**advrmsl_sd** (:) *[real,private/allocatable]*
- rms/**chebt_rms** *[costf_odd_t,public]*
- rms/**diftor2hint** (:,:,:) *[real,allocatable/public]*
- rms/**lfrmssd** (:) *[real,private/allocatable]*

- rms/**dtvrmsl_sd** (:) *[real,private/allocatable]*
- rms/**prermssd** (:) *[real,private/allocatable]*
- rms/**lf2hint** (:,:) *[real,allocatable/public]*
- rms/**ciarmsl_ta** (:) *[real,private/allocatable]*
- rms/**buormssd** (:) *[real,private/allocatable]*
- rms/**corrmssd** (:) *[real,private/allocatable]*
- rms/**plfrmsl_ta** (:) *[real,private/allocatable]*
- rms/**magrmsl_sd** (:) *[real,private/allocatable]*
- rms/**geormsl_ta** (:) *[real,private/allocatable]*
- rms/**dtbpollmr** (:,:) *[complex,allocatable/public]*
- rms/**advrmssd** (:) *[real,private/allocatable]*
- rms/**dtvpollmr** (:,:) *[complex,allocatable/public]*
- rms/**plfrmssd** (:) *[real,private/allocatable]*
- rms/**dtvrmsl_ta** (:) *[real,private/allocatable]*
- rms/**arc2hint** (:,:) *[real,allocatable/public]*
- rms/**clfrmssd** (:) *[real,private/allocatable]*
- rms/**dtbtor2hint** (:,:,:,:) *[real,allocatable/public]*
- rms/**buormsl_ta** (:) *[real,private/allocatable]*
- rms/**clfrmsl_ta** (:) *[real,private/allocatable]*
- rms/**arcrmssd** (:) *[real,private/allocatable]*
- rms/**adv2hint** (:,:) *[real,allocatable/public]*
- rms/**clf2hint** (:,:) *[real,allocatable/public]*
- rms/**difpol2hint** (:,:,:,:) *[real,allocatable/public]*
- rms/**n_cheb_maxc** *[integer,public]*
  Number of Chebyshevs
- rms/**lfrmsl_sd** (:) *[real,private/allocatable]*
- rms/**arcrmsl_sd** (:) *[real,private/allocatable]*
- rms/**geormsl_sd** (:) *[real,private/allocatable]*
- rms/**ciarmsl_sd** (:) *[real,private/allocatable]*
- rms/**dr_facc** (:) *[real,allocatable/public]*
- rms/**dtvpol2hint** (:,:,:,:) *[real,allocatable/public]*
- rms/**difrmssd** (:) *[real,private/allocatable]*
- rms/**clfrmsl_sd** (:) *[real,private/allocatable]*
- rms/**prermsl_ta** (:) *[real,private/allocatable]*
- rms/**ciarmssd** (:) *[real,private/allocatable]*
- rms/**rc** (:) *[real,private/allocatable]*

- rms/**buo2hint** (:,:) *[real,allocatable/public]*
- rms/**difrmsl_ta** (:) *[real,private/allocatable]*
- rms/**difpollmr** (:,:) *[complex,allocatable/public]*
- rms/**prermsl_sd** (:) *[real,private/allocatable]*
- rms/**lfrmsl_ta** (:) *[real,private/allocatable]*
- rms/**cor2hint** (:,:) *[real,allocatable/public]*
- rms/**mag2hint** (:,:) *[real,allocatable/public]*
- rms/**arcrmsl_ta** (:) *[real,private/allocatable]*
- rms/**plf2hint** (:,:) *[real,allocatable/public]*
- rms/**magrmsl_ta** (:) *[real,private/allocatable]*
- rms/**pre2hint** (:,:) *[real,allocatable/public]*
- rms/**eps** *[real,private/parameter=10.0_cp*epsilon(one)]*
- rms/**magrmssd** (:) *[real,private/allocatable]*
- rms/**difrmsl_sd** (:) *[real,private/allocatable]*
- rms/**geormssd** (:) *[real,private/allocatable]*
- rms/**n_r_maxc** *[integer,public]*
  Number of radial points
- rms/**corrmsl_sd** (:) *[real,private/allocatable]*

Subroutines and functions

subroutine rms/**initialize_rms**()
Memory allocation

Called from *magic*

Call to *init_rnb()*

subroutine rms/**zerorms**()
Zeros integrals that are set in get_td, update_z, update_wp, update_b, dtVrms and dtBrms

Called from *output()*

subroutine rms/**init_rnb**(*r*, *rcut*, *rdea*, *r2*, *n_r_max2*, *n_cheb_max2*, *ns*, *dr_fac2*, *chebt_rms*, *ndi_costf1*, *ndd_costf1*)

Prepares the usage of a cut back radial grid where nS points on both boundaries are discarded. The aim actually is to discard boundary effects, but just not considering the boundary grid points does not work when you also want radial derivatives and integrals. For these we use the Chebychev transform which needs are particular number of grid points so that the fast cosine transform can be applied. Therefor more than just 2 points have to be thrown away, which may make sense anyway.

Parameters

- **r** (*n_r_max*) *[real,in]*
- **rcut** *[real,in]*
- **rdea** *[real,in]*

- **r2** (*) *[real,out,allocatable]*
- **n_r_max2** *[integer,out]*
- **n_cheb_max2** *[integer,out]*
- **ns** *[integer,out]*
- **dr_fac2** (*) *[real,out,allocatable]*
- **chebt_rms** *[costf_odd_t,out]*
- **ndi_costf1** *[integer,in]*
- **ndd_costf1** *[integer,in]*

Called from *initialize_rms()*

Call to *cheb_grid()*

subroutine rms/**dtvrms** (*time*, *nrms_sets*, *timepassed*, *timenorm*)

This routine calculates and stores the different contributions of the forces entering the Navier-Stokes equation.

Parameters

- **time** *[real,in]*
- **nrms_sets** *[integer,inout]*
- **timepassed** *[real,in]*
- **timenorm** *[real,in]*

Called from *output()*

Call to *myallgather()*, *rint_r()*, *getmsd2()*, *get_drns()*, *hint2dpol()*

subroutine rms/**dtbrms** (*time*)

Parameters **time** *[real,in]*

Called from *output()*

Call to *myallgather()*, *get_drns()*, *get_poltorrms()*, *hint2dpollm()*, *rint()*, *get_ras()*, *get_paslm()*

10.17.2 RMS_helpers.f90

Description

This module contains several useful subroutines required to compute RMS diagnostics

Quick access

Routines *get_ras()*, *hint2dpollm()*, *hint2dpol()*, *hintrms()*, *get_poltorrms()*, *hint2pollm()*, *hint2pol()*, *get_paslm()*, *hint2torlm()*, *hint2tor()*

Needed modules

- *horizontal_data* (*osn1()*, *plm()*, *dlh()*, *dplm()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *useful* (*cc2real()*): library with several useful subroutines
- *integration* (*rint_r()*): Radial integration functions
- *precision_mod*: This module controls the precision used in MagIC
- *truncation* (*lm_max()*, *l_max()*, *n_r_max()*, *lm_max_dtb()*): This module defines the grid points and the truncation
- *lmmapping* (mappings())
- *blocking* (*lm2()*, *st_map()*): Module containing blocking information
- *constants* (*vol_oc()*, *one()*): module containing constants and parameters used in the code.
- *radial_functions* (*drx()*, *r()*, *or2()*, *chebt_oc()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

Subroutines and functions

subroutine rms_helpers/**get_paslm**(*tlm*, *bp*, *rt*, *nthetastart*, *sizethetablock*)

Purpose of this subroutine is to calculated the axisymmetric phi component Bp of an axisymmetric toroidal field Tlm given in spherical harmonic space (l,m=0).

Parameters

- **tlm** (*lm_max_dtb*) *[complex,in]* :: field in (l,m)-space for rT
- **bp** (*) *[real,out]*
- **rt** *[real,in]* :: radius
- **nthetastart** *[integer,in]* :: first theta to be treated
- **sizethetablock** *[integer,in]* :: size of theta block

Called from *dtbrms()*

subroutine rms_helpers/**get_poltorrms**(*pol*, *drpol*, *tor*, *polrms*, *torrms*, *polasrms*, *torasrms*, *map*)

calculates integral PolRms=sqrt(Integral (pol^2 dV)) calculates integral TorRms=sqrt(Integral (tor^2 dV)) plus axisymmetric parts. integration in theta,phi by summation of spherical harmonics integration in r by using Chebycheff integrals The mapping map gives the mapping lm to l,m for the input arrays Pol,drPol and Tor Output: PolRms,TorRms,PolAsRms,TorAsRms

Parameters

- **pol** (*lm_max*,*n_r_max*) *[complex,in]* :: Poloidal field Potential
- **drpol** (*lm_max*,*n_r_max*) *[complex,in]* :: Radial derivative of Pol
- **tor** (*lm_max*,*n_r_max*) *[complex,in]* :: Toroidal field Potential
- **polrms** *[real,out]*

- **torrms** *[real,out]*
- **polasrms** *[real,out]*
- **torasrms** *[real,out]*
- **map** *[mappings,in]*

Called from *dtbrms()*

Call to *cc2real()*, *rint_r()*

subroutine rms_helpers/**hint2dpol**(*dpol*, *lmstart*, *lmstop*, *pol2hint*, *map*)

Parameters

- **dpol** (*lm_max*) *[complex,in]* :: Toroidal field Potential
- **lmstart** *[integer,in]*
- **lmstop** *[integer,in]*
- **pol2hint** (*l_max*)+1) *[real,inout]*
- **map** *[mappings,in]*

Called from *dtvrms()*

Call to *cc2real()*

subroutine rms_helpers/**hint2dpollm**(*dpol*, *lmstart*, *lmstop*, *pol2hint*, *map*)

Parameters

- **dpol** (*lm_max*) *[complex,in]*
- **lmstart** *[integer,in]*
- **lmstop** *[integer,in]*
- **pol2hint** (*lm_max*) *[real,inout]*
- **map** *[mappings,in]*

Called from *dtbrms()*

Call to *cc2real()*

subroutine rms_helpers/**hint2pol**(*pol*, *lb*, *ub*, *nr*, *lmstart*, *lmstop*, *pollmr*, *pol2hint*, *map*)

Parameters

- **pol** (ub-lb+1) *[complex,in]*
- **lb** *[integer,in]*
- **ub** *[integer,in]* :: Poloidal field Potential
- **nr** *[integer,in]*
- **lmstart** *[integer,in]*
- **lmstop** *[integer,in]*
- **pollmr** (*lm_max*) *[complex,out]*
- **pol2hint** (*l_max*)+1) *[real,inout]*
- **map** *[mappings,in]*

Called from *updatewp()*

Call to *cc2real()*

subroutine rms_helpers/**hint2pollm**(*pol*, *lb*, *ub*, *nr*, *lmstart*, *lmstop*, *pollmr*, *pol2hint*, *map*)

Parameters

- **pol** (ub-lb+1) *[complex,in]*
- **lb** *[integer,in]*
- **ub** *[integer,in]* :: Poloidal field Potential
- **nr** *[integer,in]*
- **lmstart** *[integer,in]*
- **lmstop** *[integer,in]*
- **pollmr** (*lm_max*) *[complex,out]*
- **pol2hint** (*lm_max*) *[real,inout]*
- **map** *[mappings,in]*

Called from *updateb()*

Call to *cc2real()*

subroutine rms_helpers/**hintrms**(*f*, *nr*, *lmstart*, *lmstop*, *lmp*, *f2hint*, *map*)

Parameters

- **f** (*) *[complex,in]*
- **nr** *[integer,in]*
- **lmstart** *[integer,in]*
- **lmstop** *[integer,in]*
- **lmp** *[integer,in]*
- **f2hint** (*l_max*)+1) *[real,inout]*
- **map** *[mappings,in]*

Called from *get_td()*

Call to *cc2real()*

subroutine rms_helpers/**hint2tor**(*tor*, *lb*, *ub*, *nr*, *lmstart*, *lmstop*, *tor2hint*, *map*)

Parameters

- **tor** (ub-lb+1) *[complex,in]*
- **lb** *[integer,in]*
- **ub** *[integer,in]* :: Toroidal field Potential
- **nr** *[integer,in]*
- **lmstart** *[integer,in]*
- **lmstop** *[integer,in]*
- **tor2hint** (*l_max*)+1) *[real,inout]*
- **map** *[mappings,in]*

Called from *updatez()*

Call to *cc2real()*

subroutine rms_helpers/**hint2torlm**(*tor*, *lb*, *ub*, *nr*, *lmstart*, *lmstop*, *tor2hint*, *map*)

Parameters

- **tor** (ub-lb+1) *[complex,in]*
- **lb** *[integer,in]*
- **ub** *[integer,in]* :: Toroidal field Potential
- **nr** *[integer,in]*
- **lmstart** *[integer,in]*
- **lmstop** *[integer,in]*
- **tor2hint** (*lm_max*) *[real,inout]*
- **map** *[mappings,in]*

Called from *updateb()*

Call to *cc2real()*

subroutine rms_helpers/**get_ras**(*blm*, *br*, *rt*, *nthetastart*, *sizethetablock*)

Purpose of this subroutine is to calculate the axisymmetric radial component Br of an axisymmetric ploidal field Blm given in spherical harmonic space (l,m=0).

Parameters

- **blm** (*lm_max_dtb*) *[complex,in]* :: field in (l,m)-space for rT
- **br** (*) *[real,out]*
- **rt** *[real,in]* :: radius
- **nthetastart** *[integer,in]* :: first theta to be treated
- **sizethetablock** *[integer,in]* :: last theta

Called from *dtbrms()*

10.17.3 dtB.f90

Description

This module contains magnetic field stretching and advection terms plus a separate omega-effect. It is used for movie output.

Quick access

Variables *padvasrms*, *padvrms*, *pdifasrms*, *pdifrms*, *pstrasrms*, *pstrrms*, *tadvasrms*, *tadvrms*, *tdifasrms*, *tdifrms*, *tomeasrms*, *tomerms*, *tstrasrms*, *tstrrms*, *padvlm*, *padvlm_rloc*, *padvlmic*, *padvlmic_lmloc*, *pdiflm*, *pdiflm_lmloc*, *pdiflmic*, *pdiflmic_lmloc*, *pstrlm*, *pstrlm_rloc*, *tadvlm*, *tadvlm_rloc*, *tadvlmic*, *tadvlmic_lmloc*, *tadvrlm*, *tadvrlm_rloc*, *tdiflm*, *tdiflm_lmloc*, *tdiflmic*, *tdiflmic_lmloc*, *tomelm*, *tomelm_rloc*, *tomerlm*, *tomerlm_rloc*, *tstrlm*, *tstrlm_rloc*, *tstrrlm*, *tstrrlm_rloc*

Routines *dtb_gather_rloc_on_rank0()*, *initialize_dtb_mod()*, *get_dh_dtblm()*, *get_dtblm()*, *get_dtblmfinish()*

Needed modules

- *horizontal_data* (*hdif_b()*, *osn1()*, *osn2()*, *dlh()*, *cosn2()*, *d_lp1()*, *dtheta1s()*, *dphi()*, *dtheta1a()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *parallel_mod*: This module contains the blocking information
- *fft*: This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5
- *lmloop_data* (*ulm()*, *llmmag()*, *llm()*, *ulmmag()*)
- *radial_spectra*
- *radial_data* (*nrstart()*, *nrstop()*)
- *precision_mod*: This module controls the precision used in MagIC
- *communications* (*gather_all_from_lo_to_rank0()*, *gt_ic()*, *gt_oc()*)
- *legendre_grid_to_spec* (*legtf2()*, *legtf3()*)
- *logic* (*l_cond_ic()*, *l_dtrmagspec()*): Module containing the logicals that control the run
- *radial_der* (*get_drns()*): Radial derivatives functions
- *physical_parameters* (*opm()*, *o_sr()*): Module containing the physical parameters
- *truncation* (*n_r_ic_max()*, *l_max()*, *n_r_ic_maxmag()*, *n_r_ic_max_dtb()*, *ldtbmem()*, *n_phi_max()*, *n_r_max_dtb()*, *n_r_max()*, *lm_max_dtb()*, *lm_max()*, *nrp()*, *n_cheb_max()*, *n_r_maxmag()*): This module defines the grid points and the truncation
- *blocking* (*l2lmas()*, *lm2lmp()*, *lm2m()*, *lm2l()*, *nfs()*, *lmp2lmps()*, *lo_map()*, *lmp2lmpa()*, *st_map()*): Module containing blocking information
- *constants* (*two()*): module containing constants and parameters used in the code.
- *radial_functions* (*drx()*, *orho1()*, *chebt_oc()*, *or2()*, *or1()*, *dllambda()*, *o_r_ic()*, *lambda()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

- dtb_mod/**tadvlmic** (:,:) *[complex,allocatable/public]*
- dtb_mod/**pdifrms** *[real,public]*
- dtb_mod/**pdifasrms** *[real,public]*
- dtb_mod/**tadvlmic_lmloc** (:,:) *[complex,allocatable/public]*
- dtb_mod/**tadvlm_rloc** (:,:) *[complex,allocatable/public]*
- dtb_mod/**padvlm** (:,:) *[complex,allocatable/public]*
- dtb_mod/**padvasrms** *[real,public]*
- dtb_mod/**tdifrms** *[real,public]*
- dtb_mod/**tstrlm** (:,:) *[complex,allocatable/public]*

- dtb_mod/**pdiflm_lmloc** (:,:) *[complex,allocatable/public]*
- dtb_mod/**tdiflm** (:,:) *[complex,allocatable/public]*
- dtb_mod/**tdiflmic** (:,:) *[complex,allocatable/public]*
- dtb_mod/**tomerms** *[real,public]*
- dtb_mod/**tstrrms** *[real,public]*
- dtb_mod/**tadvlm** (:,:) *[complex,allocatable/public]*
- dtb_mod/**pstrasrms** *[real,public]*
- dtb_mod/**tdiflm_lmloc** (:,:) *[complex,allocatable/public]*
- dtb_mod/**tstrlm_rloc** (:,:) *[complex,allocatable/public]*
- dtb_mod/**tadvrlm_rloc** (:,:) *[complex,allocatable/public]*
- dtb_mod/**padvlmic** (:,:) *[complex,allocatable/public]*
- dtb_mod/**pdiflm** (:,:) *[complex,allocatable/public]*
- dtb_mod/**tdiflmic_lmloc** (:,:) *[complex,allocatable/public]*
- dtb_mod/**pdiflmic** (:,:) *[complex,allocatable/public]*
- dtb_mod/**tstrasrms** *[real,public]*
- dtb_mod/**pdiflmic_lmloc** (:,:) *[complex,allocatable/public]*
- dtb_mod/**tadvasrms** *[real,public]*
- dtb_mod/**pstrlm_rloc** (:,:) *[complex,allocatable/public]*
- dtb_mod/**tstrrlm_rloc** (:,:) *[complex,allocatable/public]*
- dtb_mod/**tomelm_rloc** (:,:) *[complex,allocatable/public]*
- dtb_mod/**tadvrms** *[real,public]*
- dtb_mod/**pstrrms** *[real,public]*
- dtb_mod/**tomerlm** (:,:) *[complex,allocatable/public]*
- dtb_mod/**padvlmic_lmloc** (:,:) *[complex,allocatable/public]*
- dtb_mod/**padvrms** *[real,public]*
- dtb_mod/**tdifasrms** *[real,public]*
- dtb_mod/**tomeasrms** *[real,public]*
- dtb_mod/**tomerlm_rloc** (:,:) *[complex,allocatable/public]*
- dtb_mod/**pstrlm** (:,:) *[complex,allocatable/public]*
- dtb_mod/**padvlm_rloc** (:,:) *[complex,allocatable/public]*
- dtb_mod/**tadvrlm** (:,:) *[complex,allocatable/public]*
- dtb_mod/**tomelm** (:,:) *[complex,allocatable/public]*
- dtb_mod/**tstrrlm** (:,:) *[complex,allocatable/public]*

Subroutines and functions

subroutine dtb_mod/**initialize_dtb_mod**()
Memory allocation

Called from *magic*

subroutine dtb_mod/**dtb_gather_rloc_on_rank0**()
MPI communicators for dtB outputs

Called from *get_dtblmfinish()*

subroutine dtb_mod/**get_dtblm**(*nr*, *vr*, *vt*, *vp*, *br*, *bt*, *bp*, *n_theta_start*, *n_theta_block*, *btvrlm*, *bpvrlm*, *brvtlm*, *brvplm*, *btvplm*, *bpvtlm*, *brvzlm*, *btvzlm*, *btvpcotlm*, *bpvtcotlm*, *btvzcotlm*, *btvpsn2lm*, *bpvtsn2lm*, *btvzsn2lm*)

Parameters

- **nr** *[integer,in]*
- **vr** (*nrp*,*nfs*) *[real,in]*
- **vt** (*nrp*,*nfs*) *[real,in]*
- **vp** (*nrp*,*nfs*) *[real,in]*
- **br** (*nrp*,*nfs*) *[real,in]*
- **bt** (*nrp*,*nfs*) *[real,in]*
- **bp** (*nrp*,*nfs*) *[real,in]*
- **n_theta_start** *[integer,in]*
- **n_theta_block** *[integer,in]*
- **btvrlm** (*) *[complex,out]*
- **bpvrlm** (*) *[complex,out]*
- **brvtlm** (*) *[complex,out]*
- **brvplm** (*) *[complex,out]*
- **btvplm** (*) *[complex,out]*
- **bpvtlm** (*) *[complex,out]*
- **brvzlm** (*) *[complex,out]*
- **btvzlm** (*) *[complex,out]*
- **btvpcotlm** (*) *[complex,out]*
- **bpvtcotlm** (*) *[complex,out]*
- **btvzcotlm** (*) *[complex,out]*
- **btvpsn2lm** (*) *[complex,out]*
- **bpvtsn2lm** (*) *[complex,out]*
- **btvzsn2lm** (*) *[complex,out]*

Called from *do_iteration_thetablocking_openmp()*,do_iteration_thetablocking_seq(), do_iteration_thetablocking_shtns()

Call to *fft_thetab()*, *legtf3()*, *legtf2()*

subroutine dtb_mod/**get_dtblmfinish**(*time*, *n_time_step*, *omega_ic*, *b*, *ddb*, *aj*, *dj*, *ddj*, *b_ic*, *db_ic*, *ddb_ic*, *aj_ic*, *dj_ic*, *ddj_ic*)

Parameters

- **time** *[real,in]*
- **n_time_step** *[integer,in]*
- **omega_ic** *[real,in]*
- **b** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,in]*
- **ddb** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,in]*
- **aj** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,in]*
- **dj** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,in]*
- **ddj** (*ulmmag*-(*llmmag*)+1,*n_r_maxmag*) *[complex,in]*
- **b_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,in]*
- **db_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,in]*
- **ddb_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,in]*
- **aj_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,in]*
- **dj_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,in]*
- **ddj_ic** (*ulmmag*-(*llmmag*)+1,*n_r_ic_maxmag*) *[complex,in]*

Called from *output()*

Call to *dtb_gather_rloc_on_rank0()*, *get_drns()*, *gather_all_from_lo_to_rank0()*, *rbrspec()*, *rbpspec()*

subroutine dtb_mod/**get_dh_dtblm**(*nr*, *btvrlm*, *bpvrlm*, *brvtlm*, *brvplm*, *btvplm*, *bpvtlm*, *brvzlm*, *btvzlm*, *btvpcotlm*, *bpvtcotlm*, *btvpsn2lm*, *bpvtsn2lm*)

Purpose of this routine is to calculate theta and phi derivative related terms of the magnetic production and advection terms and store them.

Parameters

- **nr** *[integer,in]*
- **btvrlm** (*) *[complex,in]*
- **bpvrlm** (*) *[complex,in]*
- **brvtlm** (*) *[complex,in]*
- **brvplm** (*) *[complex,in]*
- **btvplm** (*) *[complex,in]*
- **bpvtlm** (*) *[complex,in]*
- **brvzlm** (*) *[complex,in]*
- **btvzlm** (*) *[complex,in]*
- **btvpcotlm** (*) *[complex,in]*
- **bpvtcotlm** (*) *[complex,in]*
- **btvpsn2lm** (*) *[complex,in]*

- **bpvtsn2lm** (*) *[complex,in]*

Called from *do_iteration_thetablocking_openmp()*, do_iteration_thetablocking_seq(), do_iteration_thetablocking_shtns()

10.17.4 dtB_arrays.f90

Needed modules

- *precision_mod* (*cp()*): This module controls the precision used in MagIC
- *constants* (*zero()*): module containing constants and parameters used in the code.
- *truncation* (*lmp_max_dtb()*): This module defines the grid points and the truncation

Types

- **type** dtb_arrays_mod/**unknown_type**

  Type fields

  - **% bpvtsn2lm** (*) *[complex,allocatable]*
  - **% btvrlm** (*) *[complex,allocatable]*
  - **% brvzlm** (*) *[complex,allocatable]*
  - **% btvzlm** (*) *[complex,allocatable]*
  - **% btvpcotlm** (*) *[complex,allocatable]*
  - **% btvpsn2lm** (*) *[complex,allocatable]*
  - **% btvzsn2lm** (*) *[complex,allocatable]*
  - **% bpvtcotlm** (*) *[complex,allocatable]*
  - **% bpvtlm** (*) *[complex,allocatable]*
  - **% brvplm** (*) *[complex,allocatable]*
  - **% bpvrlm** (*) *[complex,allocatable]*
  - **% btvzcotlm** (*) *[complex,allocatable]*
  - **% brvtlm** (*) *[complex,allocatable]*
  - **% btvplm** (*) *[complex,allocatable]*

Subroutines and functions

subroutine dtb_arrays_mod/**initialize**(*this*)

Parameters **this** *[real]*

subroutine dtb_arrays_mod/**finalize**(*this*)

Parameters **this** *[real]*

subroutine dtb_arrays_mod/**set_zero**(*this*)

Parameters **this** *[real]*

10.17.5 out_dtB_frame.f90

Quick access

Routines *lm2pt()*, *get_dtb()*, *write_dtb_frame()*, *get_bpol()*, *get_btor()*

Needed modules

- *horizontal_data* (*dplm()*, *osn1()*, *dlh()*, *d_lp1()*, *dphi()*, *n_theta_cal2ord()*, *plm()*, *sintheta()*, *costheta()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *movie_data* (*n_movie_fields_ic()*, *n_movie_field_type()*, *n_movie_surface()*, *movie_const()*, *n_movie_file()*, *n_movie_fields()*, *n_movie_const()*, *n_movie_type()*)
- *fft*: This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5
- *dtb_mod* (*pstrlm()*, *tdiflmic()*, *tadvlm()*, *tomelm()*, *tadvlmic()*, *tdiflm()*, *pdiflmic()*, *padvlm()*, *pdiflm()*, *padvlmic()*, *tstrlm()*): This module contains magnetic field stretching and advection terms plus a separate omega-effect. It is used for movie output....
- *precision_mod*: This module controls the precision used in MagIC
- *radial_der_even* (*get_drns_even()*)
- *logic* (*l_cond_ic()*): Module containing the logicals that control the run
- *radial_der* (*get_drns()*): Radial derivatives functions
- *truncation*: This module defines the grid points and the truncation
- *blocking* (*lm2()*, *nthetabs()*, *sizethetab()*, *lm2m()*, *lm2l()*, *nfs()*): Module containing blocking information
- *constants* (*zero()*, *ci()*, *one()*): module containing constants and parameters used in the code.
- *radial_functions* (*drx()*, *chebt_oc()*, *chebt_ic_even()*, *or1()*, *r_ic()*, *r()*, *dr_fac_ic()*, *r_icb()*, *chebt_ic()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

Subroutines and functions

subroutine out_dtb_frame/**write_dtb_frame**(*n_movie*, *b*, *db*, *aj*, *dj*, *b_ic*, *db_ic*, *aj_ic*, *dj_ic*)

Controls output of specific movie frames related to magnetic field production and diffusion.

Parameters

- **n_movie** *[integer,in]*
- **b** (*lm_maxmag*,*n_r_maxmag*) *[complex,in]*
- **db** (*lm_maxmag*,*n_r_maxmag*) *[complex,in]*
- **aj** (*lm_maxmag*,*n_r_maxmag*) *[complex,in]*
- **dj** (*lm_maxmag*,*n_r_maxmag*) *[complex,in]*

- **b_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,in]*
- **db_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,in]*
- **aj_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,in]*
- **dj_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,in]*

Called from *write_movie_frame()*

Call to *get_dtb()*, *get_drns()*, *get_bpol()*, *get_btor()*, *lm2pt()*, *get_drns_even()*

subroutine out_dtb_frame/**get_dtb** (*dtb*, *dtblm*, *dimb1*, *dimb2*, *n_r*, *n_theta_start*, *n_theta_block*, *l_ic*)

Parameters

- **dtb** (*) *[real,out]*
- **dtblm** (dimb1,dimb2) *[complex,in]*
- **dimb1** *[integer,in,]*
- **dimb2** *[integer,in,]*
- **n_r** *[integer,in]* :: No. of radial grid point
- **n_theta_start** *[integer,in]* :: No. of theta to start with
- **n_theta_block** *[integer,in]* :: Size of theta block
- **l_ic** *[logical,in]* :: =true if inner core field

Called from *write_dtb_frame()*

subroutine out_dtb_frame/**get_bpol** (*pollm*, *dpollm*, *br*, *bt*, *bp*, *rt*, *n_theta_start*, *n_theta_block*, *lic*)

Purpose of this subroutine is to calculate the components Br, Bt, and Bp of the poloidal magnetic field PolLM (l,m space) at the radial grid point r=rT and the block of theta grid points from n_theta=n_theta_start to n_theta=n_theta_start+n_theta_block-1 and for all phis. For lIC=.true. the inner core field is calculated, to get the IC field for a conducting inner core PolLM has to be the poloidal field at the ICB.

Parameters

- **pollm** (*lm_max*) *[complex,in]* :: field in (l,m)-space for rT
- **dpollm** (*lm_max*) *[complex,in]* :: dr field in (l,m)-space for rT
- **br** (*nrp*,*) *[real,out]*
- **bt** (*nrp*,*) *[real,out]*
- **bp** (*nrp*,*) *[real,out]*
- **rt** *[real,in]* :: radius
- **n_theta_start** *[integer,in]* :: first theta to be treated
- **n_theta_block** *[integer,in]* :: last theta
- **lic** *[logical,in]* :: true for inner core, special rDep !

Called from *write_dtb_frame()*

Call to *fft_thetab()*

subroutine out_dtb_frame/**get_btor** (*tlm*, *bt*, *bp*, *rt*, *n_theta_start*, *n_theta_block*, *lic*)

Purpose of this subroutine is to calculate the components Bt and Bp of the toroidal magnetic field Tlm (in l,m space) at the radial grid point r=rT and the block of theta grid points from n_theta=n_theta_start to n_theta=n_theta_start+n_theta_block-1 and for all phis. For lIC=.true. the inner core field is calculated, to get the IC field for a conducting inner core Plm has to be the toroidal field at the ICB.

Parameters

- **tlm** (*lm_max*) *[complex,in]* :: field in (l,m)-space for rT
- **bt** (*nrp*,*) *[real,out]*
- **bp** (*nrp*,*) *[real,out]*
- **rt** *[real,in]* :: radius
- **n_theta_start** *[integer,in]* :: first theta to be treated
- **n_theta_block** *[integer,in]* :: last theta
- **lic** *[logical,in]* :: true for inner core, special rDep !

Called from *write_dtb_frame()*

Call to *fft_thetab()*

subroutine out_dtb_frame/**lm2pt**(*alm*, *aij*, *rt*, *nthetastart*, *lic*, *lrcomp*)

Spherical harmonic transform from alm(l,m) to aij(phi,theta) Radial field components are calculated for lrComp=.true. Done within the range [n_theta_min,n_theta_min+n_theta_block-1] Used only for graphic output.

Parameters

- **alm** (*) *[complex,in]* :: field in (l,m)-space
- **aij** (*nrp*,*) *[real,out]* :: field in (theta,phi)-space
- **rt** *[real,in]*
- **nthetastart** *[integer,in]* :: first theta to be treated
- **lic** *[logical,in]* :: true for inner core, extra factor !
- **lrcomp** *[logical,in]* :: true for radial field components

Called from *write_dtb_frame()*

Call to *fft_thetab()*

10.17.6 TO.f90

Description

This module contains information for TO calculation and output

Quick access

Variables *bpsdas*, *bpsdas_rloc*, *bpzas*, *bpzas_rloc*, *bpzdas*, *bpzdas_rloc*, *bs2as*, *bs2as_rloc*, *bspas*, *bspas_rloc*, *bspdas*, *bspdas_rloc*, *bszas*, *bszas_rloc*, *bzpdas*, *bzpdas_rloc*, *ddzasl*, *dzastrlmr*, *dzastrlmr_rloc*, *dzcorlmr*, *dzcorlmr_rloc*, *dzddvplmr*, *dzddvplmr_rloc*, *dzdvplmr*, *dzdvplmr_rloc*, *dzlflmr*, *dzlflmr_rloc*, *dzrstrlmr*, *dzrstrlmr_rloc*, *dzstrlmr*, *dzstrlmr_rloc*, *v2as*, *v2as_rloc*

Routines *initialize_to()*, *to_gather_rloc_on_rank0()*, *gettofinish()*, *getto()*, *gettonext()*

Needed modules

- *horizontal_data* (*dtheta1a()*, *dlh()*, *hdif_v()*, *dtheta1s()*, *sintheta()*, *costheta()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *precision_mod*: This module controls the precision used in MagIC
- *lmloop_data* (*llmmag()*, *ulmmag()*)
- *radial_data* (*n_r_cmb()*, *nrstart()*, *nrstop()*)
- *parallel_mod*: This module contains the blocking information
- *legendre_grid_to_spec* (*legtfas2()*)
- *logic* (*l_mag()*, *lverbose()*): Module containing the logicals that control the run
- *physical_parameters* (*kbotv()*, *ktopv()*, *corfac()*): Module containing the physical parameters
- *truncation* (*n_r_maxstr()*, *l_max()*, *nrp()*, *n_phi_maxstr()*, *n_theta_maxstr()*): This module defines the grid points and the truncation
- *blocking* (*lm2()*, *nfs()*): Module containing blocking information
- *constants* (*two()*, *one()*): module containing constants and parameters used in the code.
- *radial_functions* (*orho1()*, *or4()*, *or2()*, *or3()*, *or1()*, *beta()*, *r()*, *dbeta()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

- torsional_oscillations/**v2as** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzastrlmr_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzrstrlmr_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**v2as_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bspas** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzdvplmr** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**ddzasl** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bspdas** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bzpdas** (:,:) *[real,allocatable/public]*

- torsional_oscillations/**bspdas_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzdvplmr_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bpsdas** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzstrlmr_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzcorlmr_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bpzas** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bzpdas_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzddvplmr_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzastrlmr** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bszas** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzddvplmr** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzlflmr** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzcorlmr** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bs2as_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzrstrlmr** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bspas_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bpzas_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzlflmr_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bs2as** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzstrlmr** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bpsdas_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bpzdas_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bpzdas** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bszas_rloc** (:,:) *[real,allocatable/public]*

Subroutines and functions

subroutine torsional_oscillations/**initialize_to**()

Allocate the memory needed

Called from *magic*

subroutine torsional_oscillations/**getto**(*vr*, *vt*, *vp*, *cvr*, *dvpdr*, *br*, *bt*, *bp*, *cbr*, *cbt*, *bslast*, *bplast*, *bzlast*, *dzrstrlm*, *dzastrlm*, *dzcorlm*, *dzlflm*, *dtlast*, *nr*, *nthetastart*, *nthetablocksize*)

This program calculates various axisymmetric linear and nonlinear variables for a radial grid point nR and a theta-block. Input are the fields vr,vt,vp,cvr,dvpdr Output are linear azimuthally averaged field VpAS (flow phi component), VpAS2 (square of flow phi component), V2AS (V*V), and Coriolis force Cor. These are give in (r,theta)-space. Also in (r,theta)-space

are azimuthally averaged correlations of non-axisymmetric flow components and the respective squares: Vsp=Vs*Vp,Vzp,Vsz,VspC,VzpC,VszC. These are used to calulcate the respective correlations and Reynolds stress. In addition three output field are given in (lm,r) space: dzRstrLMr,dzAstrLMr,dzCorLM,dzLFLM.

These are used to calculate the total Reynolds stress, advection and viscous stress later. Their calculation retraces the calculations done in the time-stepping part of the code.

Parameters

- **vr** (*nrp*,*nfs*) *[real,in]*
- **vt** (*nrp*,*nfs*) *[real,in]*
- **vp** (*nrp*,*nfs*) *[real,in]*
- **cvr** (*nrp*,*nfs*) *[real,in]*
- **dvpdr** (*nrp*,*nfs*) *[real,in]*
- **br** (*nrp*,*nfs*) *[real,in]*
- **bt** (*nrp*,*nfs*) *[real,in]*
- **bp** (*nrp*,*nfs*) *[real,in]*
- **cbr** (*nrp*,*nfs*) *[real,in]*
- **cbt** (*nrp*,*nfs*) *[real,in]*
- **bslast** (*n_phi_maxstr*,*n_theta_maxstr*,*nrstop*-(*nrstart*)+1) *[real,in]*
- **bplast** (*n_phi_maxstr*,*n_theta_maxstr*,*nrstop*-(*nrstart*)+1) *[real,in]*
- **bzlast** (*n_phi_maxstr*,*n_theta_maxstr*,*nrstop*-(*nrstart*)+1) *[real,in]*
- **dzrstrlm** (*l_max*+2) *[real,out]*
- **dzastrlm** (*l_max*+2) *[real,out]*
- **dzcorlm** (*l_max*+2) *[real,out]*
- **dzlflm** (*l_max*+2) *[real,out]*
- **dtlast** *[real,in]* :: last time step
- **nr** *[integer,in]* :: radial grid point
- **nthetastart** *[integer,in]* :: theta block
- **nthetablocksize** *[integer,in]*

Called from *do_iteration_thetablocking_openmp()*,`do_iteration_thetablocking_seq()`, `do_iteration_thetablocking_shtns()`

Call to *legtfas2()*

subroutine `torsional_oscillations/`**`gettonext`**(*zas*, *br*, *bt*, *bp*, *ltonext*, *ltonext2*, *dt*, *dtlast*, *nr*, *nthetastart*, *nthetablocksize*, *bslast*, *bplast*, *bzlast*)

Preparing TO calculation by storing flow and magnetic field contribution to build time derivative.

Parameters

- **zas** (*l_max*+1) *[real,in]*
- **br** (*nrp*,*nfs*) *[real,in]*

- **bt** (*nrp*,*nfs*) *[real,in]*
- **bp** (*nrp*,*nfs*) *[real,in]*
- **ltonext** *[logical,in]*
- **ltonext2** *[logical,in]*
- **dt** *[real,in]*
- **dtlast** *[real,in]*
- **nr** *[integer,in]*
- **nthetastart** *[integer,in]*
- **nthetablocksize** *[integer,in]*
- **bslast** (*n_phi_maxstr*,*n_theta_maxstr*,*nrstop*-(*nrstart*)+1) *[real,out]*
- **bplast** (*n_phi_maxstr*,*n_theta_maxstr*,*nrstop*-(*nrstart*)+1) *[real,out]*
- **bzlast** (*n_phi_maxstr*,*n_theta_maxstr*,*nrstop*-(*nrstart*)+1) *[real,out]*

Called from *do_iteration_thetablocking_openmp()*, do_iteration_thetablocking_seq(), do_iteration_thetablocking_shtns()

subroutine torsional_oscillations/**gettofinish**(*nr*, *dtlast*, *zas*, *dzas*, *ddzas*, *dzrstrlm*, *dzastrlm*, *dzcorlm*, *dzlflm*)

Parameters

- **nr** *[integer,in]*
- **dtlast** *[real,in]*
- **zas** (*l_max*+1) *[real,in]*
- **dzas** (*l_max*+1) *[real,in]*
- **ddzas** (*l_max*+1) *[real,in]*
- **dzrstrlm** (*l_max*+2) *[real,in]*
- **dzastrlm** (*l_max*+2) *[real,in]*
- **dzcorlm** (*l_max*+2) *[real,in]*
- **dzlflm** (*l_max*+2) *[real,in]*

Called from *do_iteration_thetablocking_openmp()*, do_iteration_thetablocking_seq(), do_iteration_thetablocking_shtns()

subroutine torsional_oscillations/**to_gather_rloc_on_rank0**()

MPI communicators for TO outputs

Called from *outto()*

10.17.7 TO_helpers.f90

Description

This module contains several helpful subroutines used in the TO calculations

Quick access

Routines *getpastr()*, *getastr()*, *get_pas()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *horizontal_data* (*dplm()*, *osn1()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *blocking* (*lm2()*): Module containing blocking information
- *constants* (*half()*, *two()*, *one()*): module containing constants and parameters used in the code.
- *truncation* (*l_max()*): This module defines the grid points and the truncation

Variables

Subroutines and functions

subroutine to_helpers/**getpastr** (*fz*, *flmn*, *nzmax*, *nzmaxa*, *lmmax*, *lmax*, *rmin*, *rmax*, *nchebmax*, *rz*, *dplm*, *osints*)

Parameters

- **fz** (*) *[real,out]*
- **flmn** (lmmax,*) *[real,in]*
- **nzmax** *[integer,in]*
- **nzmaxa** *[integer,in,]*
- **lmmax** *[integer,in,]*
- **lmax** *[integer,in]*
- **rmin** *[real,in]*
- **rmax** *[real,in]*
- **nchebmax** *[integer,in]*
- **rz** (0.5 * *nzmaxa* + 1.0) *[real,in]*
- **dplm** (lmmax,0.5 * *nzmaxa* + 1.0) *[real,in]*
- **osints** (0.5 * *nzmaxa* + 1.0) *[real,in]*

Called from *outpv()*, *outto()*

subroutine to_helpers/**get_pas** (*tlm*, *bp*, *rt*, *nthetastart*, *sizethetablock*)

Purpose of this subroutine is to calculate the axisymmetric phi component Bp of an axisymmetric toroidal field Tlm given in spherical harmonic space (l,m=0).

Parameters

- **tlm** (*) *[real,in]* :: field in (l,m)-space for rT
- **bp** (*) *[real,out]*

- **rt** *[real,in]* :: radius
- **nthetastart** *[integer,in]* :: first theta to be treated
- **sizethetablock** *[integer,in]* :: size of theta block

Called from *outto()*

subroutine to_helpers/**getastr** (*fz*, *flmn*, *nzmax*, *nzmaxa*, *lmmax*, *lmax*, *rmin*, *rmax*, *nchebmax*, *rz*, *plm*)

Calculates function value at radii rZ(nZmax) and colatitudes for which Plm(theta) is given from the spherical harmonic/Chebychev coefficients of an axisymmetric function (order=0).

Parameters

- **fz** (*) *[real,out]*
- **flmn** (lmmax,*) *[real,in]*
- **nzmax** *[integer,in]*
- **nzmaxa** *[integer,in,]*
- **lmmax** *[integer,in,]*
- **lmax** *[integer,in]*
- **rmin** *[real,in]*
- **rmax** *[real,in]*
- **nchebmax** *[integer,in]*
- **rz** (0.5 * *nzmaxa* + 1.0) *[real,in]*
- **plm** (lmmax,0.5 * *nzmaxa* + 1.0) *[real,in]*

Called from *outto()*

10.17.8 TO_arrays.f90

Needed modules

- *precision_mod* (*cp()*): This module controls the precision used in MagIC
- *truncation* (*l_max()*): This module defines the grid points and the truncation

Types

- **type** to_arrays_mod/**unknown_type**

  Type fields

  - **% dzcorlm** (*) *[real,allocatable]*
  - **% dzlflm** (*) *[real,allocatable]*
  - **% dzrstrlm** (*) *[real,allocatable]*
  - **% dzastrlm** (*) *[real,allocatable]*

Subroutines and functions

subroutine to_arrays_mod/**initialize**(*this*)

Parameters **this** *[real]*

subroutine to_arrays_mod/**finalize**(*this*)

Parameters **this** *[real]*

subroutine to_arrays_mod/**set_zero**(*this*)

Parameters **this** *[real]*

10.17.9 out_TO.f90

Quick access

Variables *lmmaxs*, *nzmaxs*, *chebt_z*, *osints*, *rz*, *zz*, *astrm*, *clm*, *corm*, *dvpm*, *lfm*, *rstrm*, *strm*, *vpm*, *dplms*, *plms*

Routines *initialize_outto_mod()*, *outto()*

Needed modules

- *horizontal_data* (*phi()*, *gauss()*, *theta_ord()*, *sintheta()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *chebint_mod* (*chebint()*, *chebintinit()*)
- *useful* (*logwrite()*): library with several useful subroutines
- *lmloop_data* (*llm()*, *ulm()*)
- *to_helpers* (*getpastr()*, *get_pas()*, *getastr()*): This module contains several helpful subroutines used in the TO calculations
- *legendre_grid_to_spec* (*legtfas2()*, *legtfas()*)
- *cosine_transform_odd*
- *integration* (*rint_r()*): Radial integration functions
- *parallel_mod* (rank()): This module contains the blocking information
- *num_param* (*tscale()*): Module containing numerical and control parameters
- *physical_parameters* (*pr()*, *prmag()*, *lffac()*, *ek()*, *radratio()*, *ra()*): Module containing the physical parameters
- *radial_functions* (*drx()*, *orho1()*, *r_cmb()*, *chebt_oc()*, *r()*, *r_icb()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *precision_mod*: This module controls the precision used in MagIC
- *plms_theta* (*plm_theta()*)
- *charmanip* (*dble2str()*): This module contains several useful routines to manipule character strings
- *blocking* (*st_map()*, *nthetabs()*, *nfs()*, *sizethetab()*): Module containing blocking information

- *constants* (*two()*, *one()*, *four()*, *half()*, *vol_oc()*, *pi()*): module containing constants and parameters used in the code.
- *truncation* (*l_max()*, *n_r_maxstr()*, *minc()*, *n_phi_max()*, *lstressmem()*, *n_r_max()*, *n_theta_maxstr()*, *lm_max()*, *n_theta_max()*): This module defines the grid points and the truncation
- *torsional_oscillations* (*bs2as()*, *dzstrlmr()*, *dzlflmr()*, *dzrstrlmr()*, *bpzas()*, *to_gather_rloc_on_rank0()*, *dzcorlmr()*, *bspas()*, *dzrstrlmr_rloc()*, *dzastrlmr()*, *dzdvplmr()*, *bpsdas()*, *dzddvplmr()*, *v2as()*, *bpzdas()*, *bspdas()*, *bszas()*, *bzpdas()*): This module contains information for TO calculation and output
- *output_data* (*nsmaxa()*, *n_log_file()*, *nzmaxa()*, *zdens()*, *tag()*, *runid()*, *log_file()*, *sdens()*): This module contains the parameters for output control
- *communications* (*gather_all_from_lo_to_rank0()*, *gt_oc()*)
- *logic* (*l_save_out()*, *lverbose()*): Module containing the logicals that control the run

Variables

- outto_mod/**rstrm** (:,:) *[real,private/allocatable]*
- outto_mod/**chebt_z** (:) *[costf_odd_t,private/allocatable]*
- outto_mod/**lmmaxs** *[integer,private]*
- outto_mod/**lfm** (:,:) *[real,private/allocatable]*
- outto_mod/**corm** (:,:) *[real,private/allocatable]*
- outto_mod/**astrm** (:,:) *[real,private/allocatable]*
- outto_mod/**dvpm** (:,:) *[real,private/allocatable]*
- outto_mod/**zz** (:,:) *[real,private/allocatable]*
- outto_mod/**vpm** (:,:) *[real,private/allocatable]*
- outto_mod/**nzmaxs** (:) *[integer,private/allocatable]*
- outto_mod/**clm** (:,:) *[real,private/allocatable]*
- outto_mod/**osints** (:,:) *[real,private/allocatable]*
- outto_mod/**rz** (:,:) *[real,private/allocatable]*
- outto_mod/**strm** (:,:) *[real,private/allocatable]*
- outto_mod/**dplms** (:,:,:) *[real,private/allocatable]*
- outto_mod/**plms** (:,:,:) *[real,private/allocatable]*

Subroutines and functions

subroutine outto_mod/**initialize_outto_mod**()

Called from *magic*

subroutine outto_mod/**outto**(*time*, *n_time_step*, *ekin*, *ekintas*, *noutfile*, *noutfile2*, *tofilenhs*, *tofileshs*, *movfile*, *tayfile*, *ntosets*, *ntomovsets*, *ntormssets*, *ltomov*, *ltorms*, *ltozwrite*, *z*, *omega_ic*, *omega_ma*)

Output of axisymmetric zonal flow, its relative strength, its time variation, and all forces acting on it. The slowest part in the TO process is the repetitious calculation of plms by subroutine plm_theta. They are needed in getAStr and getPAStr when I transform on the cylindrical grid. The necessary plms could simply be calculated one and then be stored for later use!

Parameters

- **time** *[real,in]*
- **n_time_step** *[integer,in]*
- **ekin** *[real,in]*
- **ekintas** *[real,in]*
- **noutfile** *[integer,in]*
- **noutfile2** *[integer,in]*
- **tofilenhs** *[character,in]*
- **tofileshs** *[character,in]*
- **movfile** *[character,in]*
- **tayfile** *[character,in]*
- **ntosets** *[integer,inout]*
- **ntomovsets** *[integer,inout]*
- **ntormssets** *[integer,inout]*
- **ltomov** *[logical,in]*
- **ltorms** *[logical,inout]*
- **ltozwrite** *[logical,inout]*
- **z** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **omega_ic** *[real,in]*
- **omega_ma** *[real,in]*

Called from *output()*

Call to *to_gather_rloc_on_rank0()*, *gather_all_from_lo_to_rank0()*, *legtfas()*, *legtfas2()*, *dble2str()*, *chebintinit()*, *plm_theta()*, *getpastr()*, *getastr()*, *chebint()*, *logwrite()*, *get_pas()*, *rint_r()*

10.17.10 radial_spectra.f90

Quick access

Routines *rbrspec()*, *rbpspec()*

Needed modules

- *horizontal_data* (*dlh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *useful* (*cc2real()*): library with several useful subroutines

- *output_data* (*tag()*): This module contains the parameters for output control
- *logic* (*l_cond_ic()*): Module containing the logicals that control the run
- *radial_data* (*n_r_icb()*)
- *precision_mod*: This module controls the precision used in MagIC
- *num_param* (*escale()*): Module containing numerical and control parameters
- *truncation* (*lm_max()*, *n_r_ic_max()*, *l_max()*, *n_r_tot()*, *n_r_max()*): This module defines the grid points and the truncation
- *lmmapping* (mappings())
- *blocking* (*st_map()*): Module containing blocking information
- *constants* (*four()*, *half()*, *pi()*, *one()*): module containing constants and parameters used in the code.
- *radial_functions* (*r_icb()*, *r_ic()*, *or2()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

Subroutines and functions

subroutine radial_spectra/**rbrspec**(*time*, *pol*, *polic*, *fileroot*, *lic*, *map*)

Parameters

- **time** *[real,in]*
- **pol** (*lm_max*,*n_r_max*) *[complex,in]*
- **polic** (*lm_max*,*n_r_ic_max*) *[complex,in]*
- **fileroot** *[character,in]*
- **lic** *[logical,in]*
- **map** *[mappings,in]*

Called from *output()*, *get_dtblmfinish()*

Call to *cc2real()*

subroutine radial_spectra/**rbpspec**(*time*, *tor*, *toric*, *fileroot*, *lic*, *map*)

Called from rank0, map gives the lm order of Tor and TorIC

Parameters

- **time** *[real,in]*
- **tor** (*lm_max*,*n_r_max*) *[complex,in]*
- **toric** (*lm_max*,*n_r_ic_max*) *[complex,in]*
- **fileroot** *[character,in]*
- **lic** *[logical,in]*
- **map** *[mappings,in]*

Called from *output()*, *get_dtblmfinish()*

Call to *cc2real()*

10.17.11 Egeos.f90

Quick access

Routines *initialize_egeos_mod()*, *getegeos()*, *getdvptr()*

Needed modules

- *horizontal_data* (*dphi()*, *phi()*, *dlh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *parallel_mod* (rank()): This module contains the blocking information
- *cosine_transform_odd*
- *fft* (*fft_to_real()*): This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5
- *chebint_mod*
- *output_data* (*nsmaxa()*, *nzmaxa()*, *zdens()*, *tag()*, *runid()*, *sdens()*): This module contains the parameters for output control
- *logic* (*l_corrmov()*, *lverbose()*, *l_anel()*): Module containing the logicals that control the run
- *lmloop_data* (*llm()*, *ulm()*)
- *precision_mod*: This module controls the precision used in MagIC
- *communications* (*gather_all_from_lo_to_rank0()*, *gt_oc()*)
- *plms_theta* (*plm_theta()*)
- *num_param* (*tscale()*): Module containing numerical and control parameters
- *physical_parameters* (*pr()*, *prmag()*, *ek()*, *g1()*, *g0()*, *radratio()*, *strat()*, *polind()*, *ra()*, *g2()*): Module containing the physical parameters
- *truncation* (*l_max()*, *minc()*, *n_phi_max()*, *nrpgeos()*, *lm_maxgeos()*, *n_r_max()*, *n_m_max()*, *m_max()*, *lm_max()*, *n_r_maxgeos()*): This module defines the grid points and the truncation
- *blocking* (*lm2m()*, *lm2l()*, *lm2mc()*): Module containing blocking information
- *constants* (*ci()*, *two()*, *one()*, *zero()*, *half()*, *pi()*): module containing constants and parameters used in the code.
- *radial_functions* (*r_icb()*, *r_cmb()*, *cheb_norm()*, *chebt_oc()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

- egeos_mod/**chebt_z** (:) *[costf_odd_t,private/allocatable]*
- egeos_mod/**osints** (:,:) *[real,private/allocatable]*
- egeos_mod/**eps** *[real,private/parameter=10.0_cp*epsilon(one)]*

- egeos_mod/**rz** (:,:) *[real,private/allocatable]*
- egeos_mod/**zz** (:,:) *[real,private/allocatable]*
- egeos_mod/**dplms** (:,:,:) *[real,private/allocatable]*
- egeos_mod/**nzmaxs** (:) *[integer,private/allocatable]*
- egeos_mod/**plms** (:,:,:) *[real,private/allocatable]*

Subroutines and functions

subroutine egeos_mod/**initialize_egeos_mod**()

Called from *magic*

subroutine egeos_mod/**getegeos**(*time*, *ngeossets*, *w*, *dw*, *ddw*, *z*, *dz*, *egeos*, *ekntc*, *ekstc*, *ekin*, *dpflow*, *dzflow*, *cvzotc*, *cvorotc*, *chelotc*)

Output of axisymmetric zonal flow, its relative strength, its time variation, and all forces acting on it. The slowest part in the TO process is the repitions calculation of Plms by subroutine plm_theta. They are needed in getDVptr when I transform on the cylindrical grid. The necessary plms could simply be calculated one and then be stored for later use! See s_outTOnew.f.

Parameters

- **time** *[real,in]*
- **ngeossets** *[integer,in]*
- **w** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **dw** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **ddw** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **z** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **dz** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **egeos** *[real,out]*
- **ekntc** *[real,out]*
- **ekstc** *[real,out]*
- **ekin** *[real,out]*
- **dpflow** *[real,out]* :: RMS length scales
- **dzflow** *[real,out]*
- **cvzotc** *[real,out]*
- **cvorotc** *[real,out]*
- **chelotc** *[real,out]*

Called from *outmisc()*

Call to *gather_all_from_lo_to_rank0()*, *chebintinit()*, *plm_theta()*, *getdvptr()*, *chebintd()*, *chebint()*

subroutine egeos_mod/**getdvptr**(*ws*, *dws*, *ddws*, *zs*, *dzs*, *rmin*, *rmax*, *rs*, *nzmax*, *nzmaxa*, *plms*, *dplms*, *osints*, *lderiv*, *vrs*, *vts*, *vps*, *vors*, *dpek*)

This subroutine calculates the three flow components VrS,VtS,VpS at a (r,theta,all phis) and (t,pi=theta, all phis). Here r=rS, PlmS=Plm(theta), dPlmS=sin(theta)*dTheta Plm(theta), and OsinTS=1/sin(theta). The flow is calculated for all n_phi_max azimuthal points used in the code, and for corresponding latitudes north and south of the equator. For lDeriv=.true. the subroutine also calculates dpEk and dzEk which are phi averages of (d Vr/d phi)**2 + (d Vtheta/ d phi)**2 + (d Vphi/ d phi)**2 and (d Vr/d z)**2 + (d Vtheta/ d z)**2 + (d Vphi/ d z)**2, respectively. These two quantities are used to calculate z and phi scale of the flow in s_getEgeos.f

Note: on input wS=w/r^2, dwS=dw/r, ddwS=ddw/r, zS=z/r

Parameters

- **ws** (*lm_max*,*n_r_max*) *[complex,in]*
- **dws** (*lm_max*,*n_r_max*) *[complex,in]*
- **ddws** (*lm_maxgeos*,*n_r_maxgeos*) *[complex,in]*
- **zs** (*lm_maxgeos*,*n_r_maxgeos*) *[complex,in]*
- **dzs** (*lm_maxgeos*,*n_r_maxgeos*) *[complex,in]*
- **rmin** *[real,in]* :: radial bounds
- **rmax** *[real,in]*
- **rs** (*nzmaxa*) *[real,in]*
- **nzmax** *[integer,in]* :: number of (r,theta) points
- **nzmaxa** *[integer,in,]*
- **plms** (*lm_maxgeos*,0.5 * *nzmaxa* + 1.0) *[real,in]*
- **dplms** (*lm_maxgeos*,0.5 * *nzmaxa* + 1.0) *[real,in]*
- **osints** (0.5 * *nzmaxa* + 1.0) *[real,in]*
- **lderiv** *[logical,in]*
- **vrs** (*nrpgeos*,*nzmaxa*) *[real,out]*
- **vts** (*nrpgeos*,*nzmaxa*) *[real,out]*
- **vps** (*nrpgeos*,*nzmaxa*) *[real,out]*
- **vors** (*nrpgeos*,*nzmaxa*) *[real,out]*
- **dpek** (*nzmaxa*) *[real,out]*

Called from `getegeos()`

Call to `fft_to_real()`

10.17.12 outPV3.f90

Quick access

Variables `dplmz`, `plmz`, `vorold`

Routines `initialize_outpv3()`, `outpv()`, `getpvptr()`

Needed modules

- *horizontal_data* (*dphi()*, *dlh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *parallel_mod* (rank()): This module contains the blocking information
- *cosine_transform_odd*
- *output_data* (*nzmaxa()*, *tag()*, *nsmaxa()*, *sdens()*): This module contains the parameters for output control
- *fft* (*fft_to_real()*): This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5
- *lmloop_data* (*llm()*, *ulm()*)
- *to_helpers* (*getpastr()*): This module contains several helpful subroutines used in the TO calculations
- *precision_mod*: This module controls the precision used in MagIC
- *communications* (*gather_all_from_lo_to_rank0()*, *gt_oc()*)
- *plms_theta* (*plm_theta()*)
- *logic* (*l_sric()*, *lverbose()*): Module containing the logicals that control the run
- *physical_parameters* (*radratio()*): Module containing the physical parameters
- *truncation* (*l_max()*, *minc()*, *n_phi_max()*, *n_r_max()*, *n_m_max()*, *m_max()*, *lm_max()*, *nrp()*): This module defines the grid points and the truncation
- *blocking* (*lm2()*, *lm2l()*, *lm2mc()*, *lm2m()*): Module containing blocking information
- *constants* (*ci()*, *two()*, *one()*, *zero()*, *half()*, *pi()*): module containing constants and parameters used in the code.
- *radial_functions* (*r_icb()*, *chebt_oc()*, *cheb_norm()*, *r_cmb()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

- outpv3/**osints** (:,:) *[real,private/allocatable]*
- outpv3/**rz** (:,:) *[real,private/allocatable]*
- outpv3/**dplmz** (:,:,:) *[real,private/allocatable]*
- outpv3/**plmz** (:,:,:) *[real,private/allocatable]*
- outpv3/**vorold** (:,:,:) *[real,private/allocatable]*
- outpv3/**dplms** (:,:,:) *[real,private/allocatable]*
- outpv3/**plms** (:,:,:) *[real,private/allocatable]*

Subroutines and functions

subroutine outpv3/**initialize_outpv3**()

Called from *magic*

subroutine outpv3/**outpv**(*time*, *l_stop_time*, *npvsets*, *w*, *dw*, *ddw*, *z*, *dz*, *omega_ic*, *omega_ma*)

Output of z-integrated axisymmetric rotation rate Vp/s and s derivatives

Parameters

- **time** *[real,in]*
- **l_stop_time** *[logical,in]*
- **npvsets** *[integer,inout]*
- **w** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **dw** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **ddw** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **z** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **dz** (*ulm*-(*llm*)+1,*n_r_max*) *[complex,in]*
- **omega_ic** *[real,in]*
- **omega_ma** *[real,in]*

Called from *output()*

Call to *gather_all_from_lo_to_rank0()*, *plm_theta()*, *getpastr()*, *getpvptr()*

subroutine outpv3/**getpvptr**(*w*, *dw*, *ddw*, *z*, *dz*, *rmin*, *rmax*, *rs*, *nzmax*, *nzmaxa*, *plms*, *dplms*, *osints*, *vrs*, *vps*, *vts*, *vors*, *dpvors*)

This subroutine calculates the three flow conponents VrS,VtS,VpS at (r,theta,all phis) and (r,pi-theta, all phis). Here r=rS, PlmS=Plm(theta), dPlmS=sin(theta)*dTheta Plm(theta), and OsinTS=1/sin(theta). The flow is calculated for all n_phi_max azimuthal points used in the code, and for corresponding latitudes north and south of the equator. For lDeriv=.true. the subroutine also calculates dpEk and dzEk which are phi averages of (d Vr/d phi)**2 + (d Vtheta/ d phi)**2 + (d Vphi/ d phi)**2 and (d Vr/d z)**2 + (d Vtheta/ d z)**2 + (d Vphi/ d z)**2, respectively. These two quantities are used ot calculate z and phi scale of the flow in s_getEgeos.f NOTE: on input w=l*(l+1)*w

Parameters

- **w** (*lm_max*,*n_r_max*) *[complex,in]*
- **dw** (*lm_max*,*n_r_max*) *[complex,in]*
- **ddw** (*lm_max*,*n_r_max*) *[complex,in]*
- **z** (*lm_max*,*n_r_max*) *[complex,in]*
- **dz** (*lm_max*,*n_r_max*) *[complex,in]*
- **rmin** *[real,in]* :: radial bounds
- **rmax** *[real,in]*
- **rs** (*nzmaxa*) *[real,in]*
- **nzmax** *[integer,in]* :: number of (r,theta) points
- **nzmaxa** *[integer,in,]*
- **plms** (*lm_max*,0.5 * *nzmaxa* + 1.0) *[real,in]*
- **dplms** (*lm_max*,0.5 * *nzmaxa* + 1.0) *[real,in]*
- **osints** (0.5 * *nzmaxa* + 1.0) *[real,in]*

- **vrs** (*nrp*,*nzmaxa*) *[real,out]*
- **vps** (*nrp*,*nzmaxa*) *[real,out]*
- **vts** (*nrp*,*nzmaxa*) *[real,out]*
- **vors** (*nrp*,*nzmaxa*) *[real,out]*
- **dpvors** (*nrp*,*nzmaxa*) *[real,out]*

Called from *outpv()*

Call to *fft_to_real()*

10.17.13 chebInt.f90

Quick access

Routines *chebintinit()*, *chebintd()*, *chebint()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *chebyshev_polynoms_mod* (*cheb_grid()*)
- *cosine_transform_odd*
- *constants* (*four()*, *two()*, *half()*): module containing constants and parameters used in the code.
- *radial_der* (*get_dcheb()*): Radial derivatives functions

Variables

Subroutines and functions

subroutine chebint_mod/**chebintinit** (*zmin*, *zmax*, *znorm*, *nnorm*, *ngridpointsmax*, *z*, *ngridpoints*, *chebt*)

Parameters

- **zmin** *[real,in]* :: integration interval !
- **zmax** *[real,in]*
- **znorm** *[real,in]* :: norm interval length
- **nnorm** *[integer,in]* :: suggested number of grid points for norm length
- **ngridpointsmax** *[integer,in]* :: dimension of z on input
- **z** (ngridpointsmax) *[real,out]* :: grid points, dimension at >= nGridPointsMax
- **ngridpoints** *[integer,out]* :: number of used grid points
- **chebt** *[costf_odd_t,out]*

Called from *getegeos()*, *outto()*

Call to *cheb_grid()*

function chebint_mod/**chebint** (*f*, *zmin*, *zmax*, *ngridpoints*, *ngridpointsmax*, *chebt*)

Parameters

- **f** (ngridpointsmax) *[real,in]* :: function on grid points
- **zmin** *[real,in]* :: integration boundaries
- **zmax** *[real,in]*
- **ngridpoints** *[integer,in]* :: No of grid points
- **ngridpointsmax** *[integer,in,]* :: No of max grid points
- **chebt** *[costf_odd_t,in]*

Return **chebint** *[real]*

Called from *getegeos()*, *outto()*

function chebint_mod/**chebintd**(*f*, *lderiv*, *zmin*, *zmax*, *ngridpoints*, *ngridpointsmax*, *chebt*)

Parameters

- **f** (ngridpointsmax) *[real,inout]* :: function on grid points
- **lderiv** *[logical,in]*
- **zmin** *[real,in]* :: integration boundaries
- **zmax** *[real,in]*
- **ngridpoints** *[integer,in]* :: No of grid points
- **ngridpointsmax** *[integer,in,]* :: No of max grid points
- **chebt** *[costf_odd_t,in]*

Return **chebintd** *[real]*

Called from *getegeos()*

10.17.14 outOmega.f90

Description

This module allows to compute the axisymmetric zonal flow versus the cylindrical radius s. By

Quick access

Variables *nsmax*

Routines *outomega()*, *lnpas2tr()*

Needed modules

- *cosine_transform_odd*
- *output_data* (*tag()*): This module contains the parameters for output control
- *precision_mod*: This module controls the precision used in MagIC
- *plms_theta* (*plm_theta()*)
- *logic* (*lverbose()*): Module containing the logicals that control the run

- *truncation* (*lm_max()*, *l_max()*, *minc()*, *n_r_max()*): This module defines the grid points and the truncation
- *blocking* (*lm2()*): Module containing blocking information
- *constants* (*half()*, *two()*, *one()*): module containing constants and parameters used in the code.
- *radial_functions* (*r_cmb()*, *r_icb()*, *chebt_oc()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

- omega/**nsmax** *[integer,private/parameter=300]*
  Number of cylindrical radial grid points

Subroutines and functions

subroutine omega/**outomega**(*z*, *omega_ic*)

Output of axisymmetric zonal flow omega(s) into field omega.TAG, where s is the cylindrical radius. This is done for the southern and norther hemispheres at z=+-(r_icb+0.5)

Parameters

- **z** (*lm_max*,*n_r_max*) *[complex,in]*
- **omega_ic** *[real,in]*

Called from *outputt()*

Call to *lnpas2tr()*

function omega/**lnpas2tr**(*f*, *lmmax*, *a*, *b*, *lmax*, *minc*, *nchebmax*, *theta*, *r*)

Parameters

- **f** (lmmax,*) *[real,in]*
- **lmmax** *[integer,in,]*
- **a** *[real,in]*
- **b** *[real,in]*
- **lmax** *[integer,in]*
- **minc** *[integer,in]*
- **nchebmax** *[integer,in]*
- **theta** *[real,in]*
- **r** *[real,in]*

Return **lnpas2tr** *[real]*

Called from *outomega()*

Call to *plm_theta()*

10.17.15 nl_special_calc.f90

Description

This module allows to calculcate several diagnostics that need to be computed in the physical space (non-linear quantities)

Quick access

Routines *get_helicity()*, *get_fluxes()*, *get_perppar()*, *get_nlblayers()*

Needed modules

- *horizontal_data* (*osn2()*, *o_sin_theta_e2()*, *sn2()*, *costheta()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *legendre_grid_to_spec* (*legtfas2()*, *legtfas()*)
- *radial_data* (*n_r_cmb()*, *n_r_icb()*)
- *precision_mod*: This module controls the precision used in MagIC
- *logic* (*l_mag_nl()*): Module containing the logicals that control the run
- *physical_parameters* (*vischeatfac()*, *ek()*): Module containing the physical parameters
- *radial_functions* (*orho1()*, *orho2()*, *or4()*, *or2()*, *or1()*, *visc()*, *beta()*, *temp0()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *blocking* (*nfs()*, *sizethetab()*): Module containing blocking information
- *constants* (*half()*, *pi()*, *two()*, *third()*, *one()*): module containing constants and parameters used in the code.
- *truncation* (*l_maxmag()*, *l_max()*, *nrp()*, *n_phi_max()*): This module defines the grid points and the truncation

Variables

Subroutines and functions

subroutine nl_special_calc/**get_nlblayers** (*vt*, *vp*, *dvtdr*, *dvpdr*, *dsdr*, *dsdt*, *dsdp*, *uhlmr*, *duhlmr*, *gradslmr*, *nr*, *nthetastart*)

Calculates axisymmetric contributions of:

•the horizontal velocity $u_h = \sqrt{u_\theta^2 + u_\phi^2}$

•its radial derivative $|\partial u_h/\partial r|$

•The thermal dissipation rate $(\nabla T)^2$

This subroutine is used when one wants to evaluate viscous and thermal dissipation layers

Parameters

- **vt** (*nrp*,*nfs*) *[real,in]*

- **vp** (*nrp*,*nfs*) *[real,in]*
- **dvtdr** (*nrp*,*nfs*) *[real,in]*
- **dvpdr** (*nrp*,*nfs*) *[real,in]*
- **dsdr** (*nrp*,*nfs*) *[real,in]*
- **dsdt** (*nrp*,*nfs*) *[real,in]*
- **dsdp** (*nrp*,*nfs*) *[real,in]*
- **uhlmr** (*l_max*+1) *[real,out]*
- **duhlmr** (*l_max*+1) *[real,out]*
- **gradslmr** (*l_max*+1) *[real,out]*
- **nr** *[integer,in]*
- **nthetastart** *[integer,in]*

Called from *do_iteration_thetablocking_openmp()*, `do_iteration_thetablocking_seq()`, `do_iteration_thetablocking_shtns()`

Call to *legtfas2()*, *legtfas()*

subroutine `nl_special_calc/`**`get_perppar`**(*vr*, *vt*, *vp*, *eperplmr*, *eparlmr*, *eperpaxilmr*, *eparaxilmr*, *nr*, *nthetastart*)

Calculates the energies parallel and perpendicular to the rotation axis

•$E_\perp = 0.5(v_s^2 + v_\phi^2)$ with $v_s = v_r \sin\theta + v_\theta \cos\theta$

•$E_\parallel = 0.5 v_z^2$ with $v_z = v_r \cos\theta - v_\theta * \sin\theta$

Parameters

- **vr** (*nrp*,*nfs*) *[real,in]*
- **vt** (*nrp*,*nfs*) *[real,in]*
- **vp** (*nrp*,*nfs*) *[real,in]*
- **eperplmr** (*l_max*+1) *[real,out]*
- **eparlmr** (*l_max*+1) *[real,out]*
- **eperpaxilmr** (*l_max*+1) *[real,out]*
- **eparaxilmr** (*l_max*+1) *[real,out]*
- **nr** *[integer,in]*
- **nthetastart** *[integer,in]*

Called from *do_iteration_thetablocking_openmp()*, `do_iteration_thetablocking_seq()`, `do_iteration_thetablocking_shtns()`

Call to *legtfas2()*

subroutine `nl_special_calc/`**`get_fluxes`**(*vr*, *vt*, *vp*, *dvrdr*, *dvtdr*, *dvpdr*, *dvrdt*, *dvrdp*, *sr*, *pr*, *br*, *bt*, *bp*, *cbt*, *cbp*, *fconvlmr*, *fkinlmr*, *fviscslmr*, *fpoynlmr*, *freslmr*, *nr*, *nthetastart*)

Calculates the fluxes:

•Convective flux: $F_c = \rho T(u_r s)$

•Kinetic flux: $F_k = 1/2\,\rho u_r(u_r^2 + u_\theta^2 + u_\phi^2)$

•Viscous flux: $F_= - (u \cdot S)_r)$

If the run is magnetic, then this routine also computes:

•Poynting flux

•resistive flux

Parameters

- **vr** (*nrp*,*nfs*) *[real,in]*
- **vt** (*nrp*,*nfs*) *[real,in]*
- **vp** (*nrp*,*nfs*) *[real,in]*
- **dvrdr** (*nrp*,*nfs*) *[real,in]*
- **dvtdr** (*nrp*,*nfs*) *[real,in]*
- **dvpdr** (*nrp*,*nfs*) *[real,in]*
- **dvrdt** (*nrp*,*nfs*) *[real,in]*
- **dvrdp** (*nrp*,*nfs*) *[real,in]*
- **sr** (*nrp*,*nfs*) *[real,in]*
- **pr** (*nrp*,*nfs*) *[real,in]*
- **br** (*nrp*,*nfs*) *[real,in]*
- **bt** (*nrp*,*nfs*) *[real,in]*
- **bp** (*nrp*,*nfs*) *[real,in]*
- **cbt** (*nrp*,*nfs*) *[real,in]*
- **cbp** (*nrp*,*nfs*) *[real,in]*
- **fconvlmr** (*l_max*+1) *[real,out]*
- **fkinlmr** (*l_max*+1) *[real,out]*
- **fvisclmr** (*l_max*+1) *[real,out]*
- **fpoynlmr** (*l_maxmag*+1) *[real,out]*
- **freslmr** (*l_maxmag*+1) *[real,out]*
- **nr** *[integer,in]*
- **nthetastart** *[integer,in]*

Called from *do_iteration_thetablocking_openmp()*, `do_iteration_thetablocking_seq()`, `do_iteration_thetablocking_shtns()`

Call to *legtfas2()*, *legtfas()*

subroutine `nl_special_calc/`**`get_helicity`**(*vr*, *vt*, *vp*, *cvr*, *dvrdt*, *dvrdp*, *dvtdr*, *dvpdr*, *hellmr*, *hel2lmr*, *helnalmr*, *helna2lmr*, *nr*, *nthetastart*)

Calculates axisymmetric contributions of helicity HelLMr and helicity**2 Hel2LMr in (l,m=0,r) space.

Parameters

- **vr** (*nrp*,*nfs*) *[real,in]*
- **vt** (*nrp*,*nfs*) *[real,in]*
- **vp** (*nrp*,*nfs*) *[real,in]*
- **cvr** (*nrp*,*nfs*) *[real,in]*
- **dvrdt** (*nrp*,*nfs*) *[real,in]*
- **dvrdp** (*nrp*,*nfs*) *[real,in]*
- **dvtdr** (*nrp*,*nfs*) *[real,in]*
- **dvpdr** (*nrp*,*nfs*) *[real,in]*
- **hellmr** (*l_max*+1) *[real,out]*
- **hel2lmr** (*l_max*+1) *[real,out]*
- **helnalmr** (*l_max*+1) *[real,out]*
- **helna2lmr** (*l_max*+1) *[real,out]*
- **nr** *[integer,in]*
- **nthetastart** *[integer,in]*

Called from *do_iteration_thetablocking_openmp()*, do_iteration_thetablocking_seq(), do_iteration_thetablocking_shtns()

Call to *legtfas2()*

10.18 Reading and storing check points (restart files)

10.18.1 readCheckPoints.f90

Description

This module contains the functions that can help reading and mapping of the restart files

Quick access

Variables *bytes_allocated*

Routines *mapdatar()*, *getlm2lmo()*, *readstartfields()*, *mapdatahydro()*, *mapdatamag()*

Needed modules

- *cosine_transform_odd*
- *radial_data* (*n_r_cmb()*, *n_r_icb()*)
- *precision_mod*: This module controls the precision used in MagIC

- *init_fields* (*tshift_ma1()*, *tshift_ma2()*, *omegaosz_ic2()*, *omegaosz_ic1()*, *n_start_file()*, *tomega_ma1()*, *tomega_ma2()*, *start_file()*, *scale_b()*, *omega_ic1()*, *omega_ic2()*, *scale_s()*, *scale_v()*, *tipdipole()*, *tomega_ic2()*, *tomega_ic1()*, *omega_ma2()*, *omega_ma1()*, *tshift_ic2()*, *tshift_ic1()*, *inform()*, *omegaosz_ma1()*, *omegaosz_ma2()*)
- *logic* (*l_rot_ic()*, *l_mag()*, *l_heat()*, *l_srma()*, *l_mag_lf()*, *l_rot_ma()*, *l_sric()*, *l_cond_ic()*): Module containing the logicals that control the run
- *physical_parameters* (*pr()*, *kbotv()*, *prmag()*, *ek()*, *sigma_ratio()*, *radratio()*, *ktopv()*, *ra()*): Module containing the physical parameters
- *truncation* (*n_r_ic_max()*, *lm_maxmag()*, *n_r_ic_maxmag()*, *minc()*, *l_max()*, *nalias()*, *n_r_max()*, *m_max()*, *lm_max()*, *lmagmem()*, *n_r_maxmag()*, *n_phi_tot()*): This module defines the grid points and the truncation
- *blocking* (*lmstartb()*, *lm2()*, *nlmbs()*, *lm2m()*, *lm2l()*, *lmstopb()*): Module containing blocking information
- *constants* (*c_z10_omega_ma()*, *pi()*, *c_z10_omega_ic()*, *zero()*, *two()*): module containing constants and parameters used in the code.
- *radial_functions* (*chebt_oc()*, *r()*, *chebt_ic()*, *cheb_norm()*, *cheb_norm_ic()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Variables

- readcheckpoints/**bytes_allocated** *[integer,private]*

Subroutines and functions

subroutine readcheckpoints/**readstartfields**(*w*, *dwdt*, *z*, *dzdt*, *p*, *dpdt*, *s*, *dsdt*, *b*, *dbdt*, *aj*, *djdt*, *b_ic*, *dbdt_ic*, *aj_ic*, *djdt_ic*, *omega_ic*, *omega_ma*, *lorentz_torque_ic*, *lorentz_torque_ma*, *time*, *dt_old*, *dt_new*, *n_time_step*)

read initial condition from restart file

Parameters

- **w** (*lm_max*,*n_r_max*) *[complex,out]*
- **dwdt** (*lm_max*,*n_r_max*) *[complex,out]*
- **z** (*lm_max*,*n_r_max*) *[complex,out]*
- **dzdt** (*lm_max*,*n_r_max*) *[complex,out]*
- **p** (*lm_max*,*n_r_max*) *[complex,out]*
- **dpdt** (*lm_max*,*n_r_max*) *[complex,out]*
- **s** (*lm_max*,*n_r_max*) *[complex,out]*
- **dsdt** (*lm_max*,*n_r_max*) *[complex,out]*
- **b** (*lm_maxmag*,*n_r_maxmag*) *[complex,out]*
- **dbdt** (*lm_maxmag*,*n_r_maxmag*) *[complex,out]*
- **aj** (*lm_maxmag*,*n_r_maxmag*) *[complex,out]*

- **djdt** (*lm_maxmag*,*n_r_maxmag*) *[complex,out]*
- **b_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,out]*
- **dbdt_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,out]*
- **aj_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,out]*
- **djdt_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,out]*
- **omega_ic** *[real,out]*
- **omega_ma** *[real,out]*
- **lorentz_torque_ic** *[real,out]*
- **lorentz_torque_ma** *[real,out]*
- **time** *[real,out]*
- **dt_old** *[real,out]*
- **dt_new** *[real,out]*
- **n_time_step** *[integer,out]*

Called from *getstartfields()*

Call to *getlm2lmo()*, *mapdatahydro()*, *mapdatamag()*

subroutine readcheckpoints/**getlm2lmo**(*n_r_max*, *n_r_max_old*, *l_max*, *l_max_old*, *m_max*, *minc*, *minc_old*, *inform*, *lm_max*, *lm_max_old*, *n_data_oldp*, *lm2lmo*)

Parameters

- **n_r_max** *[integer,in]*
- **n_r_max_old** *[integer,in]*
- **l_max** *[integer,in]*
- **l_max_old** *[integer,in]*
- **m_max** *[integer,in]*
- **minc** *[integer,in]*
- **minc_old** *[integer,in]*
- **inform** *[integer,in]*
- **lm_max** *[integer,in]*
- **lm_max_old** *[integer,out]*
- **n_data_oldp** *[integer,out]*
- **lm2lmo** (*lm_max*) *[integer,out]*

Called from *readstartfields()*

subroutine readcheckpoints/**mapdatahydro**(*wo*, *zo*, *po*, *so*, *n_data_oldp*, *lm2lmo*, *n_r_max_old*, *lm_max_old*, *n_r_maxl*, *lbc1*, *lbc2*, *lbc3*, *lbc4*, *w*, *z*, *p*, *s*)

Parameters

- **wo** (n_data_oldp) *[complex,in]*

- **zo** (n_data_oldp) *[complex,in]*
- **po** (n_data_oldp) *[complex,in]*
- **so** (n_data_oldp) *[complex,in]*
- **n_data_oldp** *[integer,in,]*
- **lm2lmo** (*lm_max*) *[integer,in]*
- **n_r_max_old** *[integer,in]*
- **lm_max_old** *[integer,in]*
- **n_r_maxl** *[integer,in]*
- **lbc1** *[logical,in]*
- **lbc2** *[logical,in]*
- **lbc3** *[logical,in]*
- **lbc4** *[logical,in]*
- **w** (*lm_max*,*n_r_max*) *[complex,out]*
- **z** (*lm_max*,*n_r_max*) *[complex,out]*
- **p** (*lm_max*,*n_r_max*) *[complex,out]*
- **s** (*lm_max*,*n_r_max*) *[complex,out]*

Called from `readstartfields()`

Call to `mapdatar()`

subroutine readcheckpoints/**mapdatamag**(*wo*, *zo*, *po*, *so*, *n_data_oldp*, *n_rad_tot*, *n_r_max_old*, *lm_max_old*, *n_r_maxl*, *lm2lmo*, *dim1*, *l_ic*, *w*, *z*, *p*, *s*)

Parameters

- **wo** (n_data_oldp) *[complex,in]*
- **zo** (n_data_oldp) *[complex,in]*
- **po** (n_data_oldp) *[complex,in]*
- **so** (n_data_oldp) *[complex,in]*
- **n_data_oldp** *[integer,in,]*
- **n_rad_tot** *[integer,in]*
- **n_r_max_old** *[integer,in]*
- **lm_max_old** *[integer,in]*
- **n_r_maxl** *[integer,in]*
- **lm2lmo** (*lm_max*) *[integer,in]*
- **dim1** *[integer,in]*
- **l_ic** *[logical,in]*
- **w** (*lm_maxmag*,dim1) *[complex,out]*
- **z** (*lm_maxmag*,dim1) *[complex,out]*
- **p** (*lm_maxmag*,dim1) *[complex,out]*
- **s** (*lm_maxmag*,dim1) *[complex,out]*

Called from *readstartfields()*

Call to *mapdatar()*

subroutine readcheckpoints/**mapdatar** (*datar*, *n_rad_tot*, *n_r_max_old*, *n_r_maxl*, *lbc*, *l_ic*)

Copy (interpolate) data (read from disc file) from old grid structure to new grid. Linear interploation is used in r if the radial grid structure differs

called in mapdata

Parameters

- **datar** (*) *[complex,out]*
- **n_rad_tot** *[integer,in]*
- **n_r_max_old** *[integer,in]*
- **n_r_maxl** *[integer,in]*
- **lbc** *[logical,in]*
- **l_ic** *[logical,in]*

Called from *mapdatamag()*, *mapdatahydro()*

10.18.2 storeCheckPoints.f90

Description

This module contains several subroutines that can be used to store the rst_#.TAG files

Quick access

Routines *store()*

Needed modules

- *fieldslast* (*d_omega_ma_dtlast()*, *d_omega_ic_dtlast()*, *lorentz_torque_iclast()*, *lorentz_torque_malast()*): This module contains time-derivaties array of the previous time-step They are needed in the time-stepping scheme.
- *output_data* (*n_rst_file()*, *rst_file()*): This module contains the parameters for output control
- *logic* (*l_heat()*, *l_mag()*, *l_cond_ic()*): Module containing the logicals that control the run
- *precision_mod*: This module controls the precision used in MagIC
- *init_fields* (*tomega_ic2()*, *tomega_ic1()*, *omegaosz_ic2()*, *omega_ic1()*, *omega_ma2()*, *omega_ma1()*, *omega_ic2()*, *inform()*, *omegaosz_ma1()*, *omegaosz_ma2()*, *tomega_ma1()*, *tomega_ma2()*, *omegaosz_ic1()*)
- *num_param* (*tscale()*): Module containing numerical and control parameters
- *physical_parameters* (*pr()*, *prmag()*, *ek()*, *sigma_ratio()*, *radratio()*, *ra()*): Module containing the physical parameters

- *truncation*(*n_r_ic_max()*, *lm_maxmag()*, *n_r_ic_maxmag()*, *minc()*, *l_max()*, *nalias()*, *n_r_max()*, *lm_max()*, *n_theta_max()*, *n_r_maxmag()*, *n_phi_tot()*): This module defines the grid points and the truncation

Variables

Subroutines and functions

subroutine storecheckpoints/**store**(*time*, *dt*, *dtnew*, *w*, *z*, *p*, *s*, *b*, *aj*, *b_ic*, *aj_ic*, *dwdtlast*, *dzdtlast*, *dpdtlast*, *dsdtlast*, *dbdtlast*, *djdtlast*, *dbdt_iclast*, *djdt_iclast*)

store results on disc file (restart file) In addition to the magnetic field and velocity potentials we store the time derivative terms djdt(lm,nR),dbdt(lm,nR), ...

Parameters

- **time** *[real,in]*
- **dt** *[real,in]*
- **dtnew** *[real,in]*
- **w** (*lm_max*,*n_r_max*) *[complex,in]*
- **z** (*lm_max*,*n_r_max*) *[complex,in]*
- **p** (*lm_max*,*n_r_max*) *[complex,in]*
- **s** (*lm_max*,*n_r_max*) *[complex,in]*
- **b** (*lm_maxmag*,*n_r_maxmag*) *[complex,in]*
- **aj** (*lm_maxmag*,*n_r_maxmag*) *[complex,in]*
- **b_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,in]*
- **aj_ic** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,in]*
- **dwdtlast** (*lm_max*,*n_r_max*) *[complex,in]*
- **dzdtlast** (*lm_max*,*n_r_max*) *[complex,in]*
- **dpdtlast** (*lm_max*,*n_r_max*) *[complex,in]*
- **dsdtlast** (*lm_max*,*n_r_max*) *[complex,in]*
- **dbdtlast** (*lm_maxmag*,*n_r_maxmag*) *[complex,in]*
- **djdtlast** (*lm_maxmag*,*n_r_maxmag*) *[complex,in]*
- **dbdt_iclast** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,in]*
- **djdt_iclast** (*lm_maxmag*,*n_r_ic_maxmag*) *[complex,in]*

Called from *output()*

10.19 Useful additional libraries

10.19.1 useful.f90

Description

library with several useful subroutines

Quick access

Routines *getmsd2()*, *logwrite()*, *factorise()*, *safeclose()*, *safeopen()*, *cc2real()*, *cc22real()*, *l_correct_step()*, *random()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod* (rank()): This module contains the blocking information
- *logic* (*l_save_out()*): Module containing the logicals that control the run
- *constants* (*one()*, *two()*, *half()*): module containing constants and parameters used in the code.
- *output_data* (*n_log_file()*, *log_file()*): This module contains the parameters for output control

Variables

Subroutines and functions

function useful/**l_correct_step**(*n*, *t*, *t_last*, *n_max*, *n_step*, *n_intervals*, *n_ts*, *times*, *n_eo*)

Parameters

- **n** *[integer,in]* :: current step
- **t** *[real,in]* :: time at current step
- **t_last** *[real,in]* :: last time at current step
- **n_max** *[integer,in]* :: max number of steps
- **n_step** *[integer,in]* :: action interval
- **n_intervals** *[integer,in]* :: number of actions
- **n_ts** *[integer,in]* :: number of times t
- **times** (*) *[real,in]* :: times where l_correct_step == true
- **n_eo** *[integer,in]* :: even/odd controller

Return **l_correct_step** *[logical]*

Called from *step_time()*

function useful/**random**(*r*)

random number generator

if (r == 0) then random(r) = next random number (between 0. and 1.)

if (r < 0) then random(r) = previous random number

if (r > 0) then

random(r) = a new sequence of random numbers is started with seed r mod 1 note: r must be a non-integer to get a different seq

called in sinit

Parameters **r** *[real,in]*

Return **random** *[real]*

Called from *initb()*, *initv()*, *inits()*

subroutine useful/**factorise**(*n*, *n_facs*, *fac*, *n_factors*, *factor*)

Purpose of this subroutine is factorize n into a number of given factors fac(i).

Parameters

- **n** *[integer,in]* :: number to be factorised !
- **n_facs** *[integer,in]* :: number of facs to be tried !
- **fac** (*) *[integer,in]* :: list of fators to be tried !
- **n_factors** *[integer,out]* :: number of factors used
- **factor** (*) *[integer,out]* :: list of factors used

Called from *init_fft()*

function useful/**cc2real**(*c*, *m*)

Parameters

- **c** *[complex,in]*
- **m** *[integer,in]*

Return **cc2real** *[real]*

Called from *hint2pol()*, *rbpspec()*, *hint2tor()*, *spectrum_average()*, *spectrum()*, *getdlm()*, *get_power()*, *hintrms()*, *hint2dpol()*, *rbrspec()*, *get_u_square()*, *spectrum_temp_average()*, *hint2dpollm()*, *outpar()*, *hint2pollm()*, *spectrum_temp()*, *getstartfields()*, *get_e_kin()*, *get_poltorrms()*, *hint2torlm()*, *get_e_mag()*

function useful/**cc22real**(*c1*, *c2*, *m*)

Parameters

- **c1** *[complex,in]*
- **c2** *[complex,in]*
- **m** *[integer,in]*

Return **cc22real** *[real]*

Called from *spectrum()*, *get_power()*, *get_e_mag()*

subroutine useful/**safeopen**(*nf*, *file_name*)

Parameters

- **nf** *[integer,in]*
- **file_name** *[character,in]*

Called from *step_time()*, *output()*, *spectrum_average()*, *spectrum_temp_average()*, *lmloop()*

subroutine useful/**safeclose**(*nf*)

Parameters **nf** *[integer,in]*

Called from *step_time()*, *output()*, *spectrum_average()*, *spectrum_temp_average()*, *lmloop()*

subroutine useful/**logwrite**(*message*)

Parameters **message** *[character,in]*

Called from *step_time()*, *initialize_movie_data()*, *output()*, *precalc()*, *dt_courant()*, *initialize_blocking()*, *getstartfields()*, *outto()*

subroutine useful/**getmsd2**(*mean*, *sd*, *x*, *n*, *dt*, *totaltime*)

This subroutine computes the mean and standard deviation according to a method introduced by Donald Knuth (1962). I rederived his formulas for a variable time step. On output SD still needs to be normalized with the totalTime and then you have to take the square root!! The input integer counts the number of calls. For n=1 initialisation is necessary.

Parameters

- **mean** *[real,out]* :: Time-average
- **sd** *[real,out]* :: Standard-deviation
- **x** *[real,in]* :: quantity to be averaged
- **n** *[integer,in]* :: number of calls(only n=1 needed)
- **dt** *[real,in]* :: time since last averaging step
- **totaltime** *[real,in]* :: total averaging time up to now

Called from *dtvrms()*

10.19.2 char_manip.f90

Description

This module contains several useful routines to manipule character strings

Quick access

Routines *dble2str()*, *capitalize()*, *str2dble()*, *delete_string()*, *length_to_blank()*, *length_to_char()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC

Subroutines and functions

subroutine charmanip/**capitalize**(*string_bn*)

Convert lower-case letters into capital letters

Parameters **string_bn** *[character,inout]*

Called from *get_movie_type()*, *step_time()*, *readnamelists()*

subroutine charmanip/**delete_string**(*string_bn*, *string_del*, *length*)

Deletes string_del from string and returns new length of string.

Parameters

- **string_bn** *[character,inout]*
- **string_del** *[character,in]*
- **length** *[integer,out]*

Called from *get_movie_type()*

subroutine charmanip/**str2dble**(*string_bn*, *num*)

interprets next word in string as an 1 real number deletes leading blanks and next_word from string

Parameters

- **string_bn** *[character,in]*
- **num** *[real,out]* :: output

Called from *get_movie_type()*

function charmanip/**length_to_blank**(*string_bn*)

determines number of characters before first blank in string

Parameters **string_bn** *[character,in]*

Return **length_to_blank** *[integer]*

Called from *openfiles()*, *get_movie_type()*, *writenamelists()*, *readnamelists()*

function charmanip/**length_to_char**(*string_bn*, *char_bn*)

Parameters

- **string_bn** *[character,in]*
- **char_bn** *[character,in]*

Return **length_to_char** *[integer]*

subroutine charmanip/**dble2str**(*num*, *str*)

converts a 1 number num into a character str

Parameters

- **num** *[real,in]*

- **str** *[character,out]*

Called from *step_time()*, *output()*, *outto()*

10.19.3 hdf5Helpers.f90

Description

This module contains several useful tools to manipulate HDF5 files

Quick access

Variables *readhdf5_attribute*, *writehdf5_attribute*

Routines *writehdf5_attr_dble()*, *writehdf5_attr_int()*, *readhdf5_attr_dble()*, *readhdf5_attr_int()*, *write_dataset()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- hdf5
- *blocking* (*lo_map()*, *st_map()*): Module containing blocking information
- *lmloop_data* (*llm()*, *ulm()*)

Variables

- hdf5helpers/**writehdf5_attribute** *[public]*
- hdf5helpers/**readhdf5_attribute** *[public]*

Subroutines and functions

subroutine hdf5helpers/**write_dataset** (*loc_id*, *dataset_name*, *dataset_type*, *dat*, *dim1*, *dims_full*)

Parameters

- **loc_id** *[integer,in]*
- **dataset_name** *[character,in]*
- **dataset_type** *[integer,in]*
- **dat** (*ulm*-(*llm*)+1,dim1) *[complex,in]*
- **dim1** *[integer,in,]*
- **dims_full** (2) *[integer,in]*

subroutine hdf5helpers/**readhdf5_attr_dble** (*loc_id*, *attr_name*, *attr_value*)

Parameters

- **loc_id** *[integer,in]*
- **attr_name** *[character,in]*

- **attr_value** *[real,out]*

subroutine hdf5helpers/**readhdf5_attr_int**(*loc_id*, *attr_name*, *attr_value*)

Parameters

- **loc_id** *[integer,in]*
- **attr_name** *[character,in]*
- **attr_value** *[integer,out]*

subroutine hdf5helpers/**writehdf5_attr_dble**(*loc_id*, *aspace_id*, *attr_name*, *attr_value*)

Parameters

- **loc_id** *[integer,in]*
- **aspace_id** *[integer,in]*
- **attr_name** *[character,in]*
- **attr_value** *[real,in]*

subroutine hdf5helpers/**writehdf5_attr_int**(*loc_id*, *aspace_id*, *attr_name*, *attr_value*)

Parameters

- **loc_id** *[integer,in]*
- **aspace_id** *[integer,in]*
- **attr_name** *[character,in]*
- **attr_value** *[integer,in]*

a

algebra, 257

b

bext, 176
blocking, 268

c

charmanip, 366
chebint_mod, 352
chebyshev_polynoms_mod, 234
communications, 181
constants, 177
cosine_transform_even, 239
cosine_transform_odd, 237
courant_mod, 203

d

dtb_arrays_mod, 333
dtb_mod, 328

e

egeos_mod, 347

f

fft, 252
fft_fac_mod, 240
fields, 164
fields_average_mod, 317
fieldslast, 167

g

general_arrays_mod, 230
getdlm_mod, 281
graphout_mod, 295
grenoble, 178

h

hdf5helpers, 368
horizontal_data, 198

i

init_fields, 189
integration, 266

k

kinetic_energy, 277

l

leg_helper_mod, 242
legendre_grid_to_spec, 250
legendre_spec_to_grid, 246
lmloop_data, 180
lmloop_mod, 208
lmmapping, 271
logic, 161

m

magnetic_energy, 279
matrices, 168
movie_data, 298

n

namelists, 186
nl_special_calc, 355
nonlinear_bcs, 232
nonlinear_lm_mod, 230
num_param, 155

o

omega, 353
out_coeff, 314
out_dtb_frame, 334
out_movie, 306
out_movie_ic, 313
outmisc_mod, 282
outpar_mod, 287
output_data, 170
output_mod, 272
outpv3, 349
outrot, 284
outto_mod, 343

p

parallel_mod, 179

physical_parameters, 157
plms_theta, 241
power, 289
precalculations, 193
precision_mod, 152

r

radial_data, 180
radial_der, 259
radial_der_even, 263
radial_functions, 195
radial_spectra, 345
radialloop, 220
readcheckpoints, 358
riteration_mod, 223
riterthetablocking_mod, 224
riterthetablocking_openmp_mod, 226
rms, 320
rms_helpers, 324

s

spectra, 291
start_fields, 187
step_time_mod, 201
store_pot_mod, 316
storecheckpoints, 362

t

timing, 205
to_arrays_mod, 342
to_helpers, 340
torsional_oscillations, 336
truncation, 153

u

updateb_mod, 217
updates_mod, 215
updatewp_mod, 209
updatez_mod, 212
useful, 364

m

magic.bLayers, 136
magic.coeff, 130
magic.cyl, 138
magic.graph2vtk, 133
magic.libmagic, 143
magic.potExtra, 135

Symbols

('general_arrays_mod/unknown_type', None) (fortran type in module general_arrays_mod), **230**
('lmmapping/allocate_mappings', None)() (fortran subroutine in module lmmapping), **272**
('lmmapping/allocate_subblocks_mappings', None)() (fortran subroutine in module lmmapping), **272**
('lmmapping/unknown_type', None) (fortran type in module lmmapping), **271**
__add__() (magic.Butterfly method), 142
__add__() (magic.Movie method), 128
__init__() (magic.AvgField method), 121
__init__() (magic.Butterfly method), 142
__init__() (magic.CompSims method), 132
__init__() (magic.MagicGraph method), 123
__init__() (magic.MagicRSpec method), 131
__init__() (magic.MagicRadial method), 121
__init__() (magic.MagicSetup method), 119
__init__() (magic.MagicSpectrum method), 122
__init__() (magic.MagicSpectrum2D method), 123
__init__() (magic.MagicTs method), 120
__init__() (magic.Movie method), 128
__init__() (magic.Movie3D method), 129
__init__() (magic.Surf method), 124
__init__() (magic.TOMovie method), 131
__init__() (magic.ThetaHeat method), 138
__init__() (magic.bLayers.BLayers method), 136
__init__() (magic.coeff.MagicCoeffCmb method), 130
__init__() (magic.coeff.MagicCoeffR method), 130
__init__() (magic.cyl.Cyl method), 139
__init__() (magic.graph2vtk.Graph2Vtk method), 134
__init__() (magic.potExtra.ExtraPot method), 135
__str__() (magic.AvgField method), 121
__str__() (magic.ThetaHeat method), 138
__str__() (magic.bLayers.BLayers method), 137

A

addtime() (fortran subroutine in module timing), **207**
adv2hint (fortran variable in module rms), **322**
advrmsl_sd (fortran variable in module rms), **321**
advrmsl_ta (fortran variable in module rms), **321**
advrmssd (fortran variable in module rms), **322**
agrav (fortran variable in module radial_functions), **196**
aj (fortran variable in module fields), **166**
aj_ave (fortran variable in module fields_average_mod), **319**
aj_ave_global (fortran variable in module fields_average_mod), **319**
aj_ic (fortran variable in module fields), **166**
aj_ic_ave (fortran variable in module fields_average_mod), **319**
aj_ic_lmloc (fortran variable in module fields), **165**
aj_lmloc (fortran variable in module fields), **165**
aj_lmloc_container (fortran variable in module fields), **165**
aj_rloc (fortran variable in module fields), **167**
aj_rloc_container (fortran variable in module fields), **166**
alffac (fortran variable in module num_param), **156**
algebra (module), **257**
allocate_common_arrays() (fortran subroutine in module riterthetablocking_mod), **225**
alph1 (fortran variable in module radial_functions), **197**
alph2 (fortran variable in module radial_functions), **196**
alpha (fortran variable in module num_param), **157**
alpha1 (fortran variable in module radial_functions), **197**
alpha2 (fortran variable in module radial_functions), **197**
amp_b1 (fortran variable in module init_fields), **190**
amp_curr (fortran variable in module bext), **176**
amp_imp (fortran variable in module bext), **176**
amp_s1 (fortran variable in module init_fields), **191**
amp_s2 (fortran variable in module init_fields), **191**
amp_v1 (fortran variable in module init_fields), **190**
amstart (fortran variable in module num_param), **156**
anelprof() (in module magic.libmagic), 143
angular_file (fortran variable in module output_data), **174**
arc2hint (fortran variable in module rms), **322**
arcrmsl_sd (fortran variable in module rms), **322**
arcrmsl_ta (fortran variable in module rms), **323**
arcrmssd (fortran variable in module rms), **322**
array_of_statuses (fortran variable in module communications), **182**
astrm (fortran variable in module outto_mod), **344**
avg() (magic.cyl.Cyl method), 139
avg() (magic.potExtra.ExtraPot method), 136

avg() (magic.Surf method), 124
AvgField (class in magic), 121
avgField() (in module magic.libmagic), 144
avgStd() (magic.Movie method), 128
avgz() (magic.cyl.Cyl method), 139

B

b (fortran variable in module fields), **166**
b0 (fortran variable in module grenoble), **178**
b_ave (fortran variable in module fields_average_mod), **319**
b_ic (fortran variable in module fields), **165**
b_ic_ave (fortran variable in module fields_average_mod), **318**
b_ic_lmloc (fortran variable in module fields), **167**
b_lmloc (fortran variable in module fields), **165**
b_lmloc_container (fortran variable in module fields), **167**
b_r_file (fortran variable in module output_data), **173**
b_rloc (fortran variable in module fields), **167**
b_rloc_container (fortran variable in module fields), **166**
beta (fortran variable in module radial_functions), **197**
bext (module), **176**
bic (fortran variable in module grenoble), **179**
BLayers (class in magic.bLayers), 136
blocking (module), **268**
bmat (fortran variable in module matrices), **169**
bmat_fac (fortran variable in module matrices), **169**
bmax_imp (fortran variable in module bext), **176**
botcond (fortran variable in module radial_functions), **197**
bots (fortran variable in module init_fields), **191**
bpeakbot (fortran variable in module init_fields), **191**
bpeaktop (fortran variable in module init_fields), **190**
bpivot (fortran variable in module matrices), **169**
bpsdas (fortran variable in module torsional_oscillations), **338**
bpsdas_rloc (fortran variable in module torsional_oscillations), **338**
bpzas (fortran variable in module torsional_oscillations), **338**
bpzas_rloc (fortran variable in module torsional_oscillations), **338**
bpzdas (fortran variable in module torsional_oscillations), **338**
bpzdas_rloc (fortran variable in module torsional_oscillations), **338**
bs2as (fortran variable in module torsional_oscillations), **338**
bs2as_rloc (fortran variable in module torsional_oscillations), **338**
bspas (fortran variable in module torsional_oscillations), **337**
bspas_rloc (fortran variable in module torsional_oscillations), **338**
bspdas (fortran variable in module torsional_oscillations), **337**
bspdas_rloc (fortran variable in module torsional_oscillations), **338**
bszas (fortran variable in module torsional_oscillations), **338**
bszas_rloc (fortran variable in module torsional_oscillations), **338**
buo (fortran variable in module updatewp_mod), **210**
buo2hint (fortran variable in module rms), **323**
buofac (fortran variable in module physical_parameters), **159**
buomeanr (fortran variable in module power), **290**
buormsl_sd (fortran variable in module rms), **321**
buormsl_ta (fortran variable in module rms), **322**
buormssd (fortran variable in module rms), **322**
Butterfly (class in magic), 142
bytes_allocated (fortran variable in module readcheckpoints), **359**
bzpdas (fortran variable in module torsional_oscillations), **337**
bzpdas_rloc (fortran variable in module torsional_oscillations), **338**

C

c_dt_z10_ic (fortran variable in module constants), **177**
c_dt_z10_ma (fortran variable in module constants), **178**
c_lorentz_ic (fortran variable in module constants), **177**
c_lorentz_ma (fortran variable in module constants), **178**
c_moi_ic (fortran variable in module constants), **177**
c_moi_ma (fortran variable in module constants), **177**
c_moi_oc (fortran variable in module constants), **177**
c_z10_omega_ic (fortran variable in module constants), **177**
c_z10_omega_ma (fortran variable in module constants), **177**
cacheblock_size_in_b (fortran variable in module blocking), **269**
capitalize() (fortran subroutine in module charmanip), **367**
cc22real() (fortran function in module useful), **365**
cc2real() (fortran function in module useful), **365**
cgesl() (fortran subroutine in module algebra), **257**
cgeslml() (fortran subroutine in module algebra), **258**
charmanip (module), **366**
cheb (fortran variable in module radial_functions), **197**
cheb_grid() (fortran subroutine in module chebyshev_polynoms_mod), **236**
cheb_ic (fortran variable in module radial_functions), **197**
cheb_int (fortran variable in module radial_functions), **197**
cheb_int_ic (fortran variable in module radial_functions), **196**

cheb_norm (fortran variable in module radial_functions), 197
cheb_norm_ic (fortran variable in module radial_functions), 197
chebgrid() (in module magic.libmagic), 144
chebint() (fortran function in module chebint_mod), 352
chebint_mod (module), 352
chebintd() (fortran function in module chebint_mod), 353
chebintinit() (fortran subroutine in module chebint_mod), 352
chebt_ic (fortran variable in module radial_functions), 197
chebt_ic_even (fortran variable in module radial_functions), 197
chebt_oc (fortran variable in module radial_functions), 196
chebt_rms (fortran variable in module rms), 321
chebt_z (fortran variable in module egeos_mod), 347
chebt_z (fortran variable in module outto_mod), 344
chebyshev_polynoms_mod (module), 234
check_time_hits() (fortran subroutine in module step_time_mod), 203
checktruncation() (fortran subroutine in module truncation), 155
chunksize (fortran variable in module parallel_mod), 179
ci (fortran variable in module constants), 177
cia2hint (fortran variable in module rms), 321
ciarmsl_sd (fortran variable in module rms), 322
ciarmsl_ta (fortran variable in module rms), 322
ciarmssd (fortran variable in module rms), 322
clf2hint (fortran variable in module rms), 322
clfrmsl_sd (fortran variable in module rms), 322
clfrmsl_ta (fortran variable in module rms), 322
clfrmssd (fortran variable in module rms), 322
clm (fortran variable in module outto_mod), 344
closefiles() (fortran subroutine in module output_data), 176
cmb_file (fortran variable in module output_data), 175
cmbhflux (fortran variable in module physical_parameters), 159
cmbmov_file (fortran variable in module output_data), 173
codeversion (fortran variable in module constants), 178
communications (module), 181
CompSims (class in magic), 132
con_decrate (fortran variable in module physical_parameters), 158
con_funcwidth (fortran variable in module physical_parameters), 159
con_lambdamatch (fortran variable in module physical_parameters), 159
con_lambdaout (fortran variable in module physical_parameters), 160
con_radratio (fortran variable in module physical_parameters), 160
conductance_ma (fortran variable in module physical_parameters), 159
constants (module), 177
cor2hint (fortran variable in module rms), 323
corfac (fortran variable in module physical_parameters), 159
corm (fortran variable in module outto_mod), 344
corrmsl_sd (fortran variable in module rms), 323
corrmsl_ta (fortran variable in module rms), 321
corrmssd (fortran variable in module rms), 322
cos36 (fortran variable in module constants), 177
cos72 (fortran variable in module constants), 178
cosine_transform_even (module), 239
cosine_transform_odd (module), 237
cosn2 (fortran variable in module horizontal_data), 200
costf1_complex() (fortran subroutine in module cosine_transform_odd), 238
costf1_complex_1d() (fortran subroutine in module cosine_transform_odd), 238
costf1_real() (fortran subroutine in module cosine_transform_odd), 238
costf1_real_1d() (fortran subroutine in module cosine_transform_odd), 238
costf2() (fortran subroutine in module cosine_transform_even), 239
costheta (fortran variable in module horizontal_data), 199
courant() (fortran subroutine in module courant_mod), 204
courant_mod (module), 203
courfac (fortran variable in module num_param), 157
cp (fortran variable in module precision_mod), 153
create_gather_type() (fortran subroutine in module communications), 184
create_lm2r_type() (fortran subroutine in module communications), 184
curlu2meanr (fortran variable in module power), 290
cut() (in module magic.libmagic), 144
Cyl (class in magic.cyl), 138
cylSder() (in module magic.libmagic), 145
cylZder() (in module magic.libmagic), 145

D

d2cheb (fortran variable in module radial_functions), 196
d2cheb_ic (fortran variable in module radial_functions), 197
d2temp0 (fortran variable in module radial_functions), 198
d3cheb (fortran variable in module radial_functions), 197
d_fft_init (fortran variable in module fft), 253
d_l (fortran variable in module horizontal_data), 200
d_lp1 (fortran variable in module horizontal_data), 200
d_m (fortran variable in module horizontal_data), 200

d_mc2m (fortran variable in module horizontal_data), 200
d_omega_ic_dtlast (fortran variable in module fieldslast), 168
d_omega_ma_dtlast (fortran variable in module fieldslast), 168
db (fortran variable in module fields), 165
db0 (fortran variable in module grenoble), 178
db_ave_global (fortran variable in module fields_average_mod), 319
db_ic (fortran variable in module fields), 166
db_ic_lmloc (fortran variable in module fields), 166
db_lmloc (fortran variable in module fields), 165
db_rloc (fortran variable in module fields), 166
dbdt_iclast (fortran variable in module fieldslast), 168
dbdt_iclast_lmloc (fortran variable in module fieldslast), 168
dbdt_lmloc (fortran variable in module step_time_mod), 202
dbdt_rloc (fortran variable in module step_time_mod), 202
dbdtlast (fortran variable in module fieldslast), 168
dbdtlast_lmloc (fortran variable in module fieldslast), 168
dbeta (fortran variable in module radial_functions), 196
dble2str() (fortran subroutine in module charmanip), 367
dcheb (fortran variable in module radial_functions), 196
dcheb_ic (fortran variable in module radial_functions), 196
ddb (fortran variable in module fields), 166
ddb0 (fortran variable in module grenoble), 178
ddb_ic (fortran variable in module fields), 165
ddb_ic_lmloc (fortran variable in module fields), 166
ddb_lmloc (fortran variable in module fields), 166
ddb_rloc (fortran variable in module fields), 166
dddrx (fortran variable in module radial_functions), 196
ddj (fortran variable in module fields), 166
ddj_ic (fortran variable in module fields), 165
ddj_ic_lmloc (fortran variable in module fields), 166
ddj_lmloc (fortran variable in module fields), 167
ddrx (fortran variable in module radial_functions), 196
ddw (fortran variable in module fields), 166
ddw_lmloc (fortran variable in module fields), 165
ddw_rloc (fortran variable in module fields), 165
ddzasl (fortran variable in module torsional_oscillations), 337
deallocate_common_arrays() (fortran subroutine in module riterthetablocking_mod), 225
defaultnamelists() (fortran subroutine in module namelists), 187
delete_string() (fortran subroutine in module charmanip), 367
delxh2 (fortran variable in module num_param), 157
delxr2 (fortran variable in module num_param), 156
dentropy0 (fortran variable in module radial_functions), 196
deriv() (in module magic.coeff), 130
destroy_gather_type() (fortran subroutine in module communications), 184
destroy_lm2r_type() (fortran subroutine in module communications), 185
dif (fortran variable in module updatewp_mod), 211
dif (fortran variable in module updatez_mod), 213
difeta (fortran variable in module num_param), 157
difexp (fortran variable in module physical_parameters), 160
difkap (fortran variable in module num_param), 156
difnu (fortran variable in module num_param), 156
difpol2hint (fortran variable in module rms), 322
difpollmr (fortran variable in module rms), 323
difrmsl_sd (fortran variable in module rms), 323
difrmsl_ta (fortran variable in module rms), 323
difrmssd (fortran variable in module rms), 322
diftor2hint (fortran variable in module rms), 321
dipcmbmean (fortran variable in module output_mod), 275
dipmean (fortran variable in module output_mod), 275
dipole_file (fortran variable in module output_data), 175
dissnb (fortran variable in module physical_parameters), 158
divktemp0 (fortran variable in module radial_functions), 197
dj (fortran variable in module fields), 165
dj_ic (fortran variable in module fields), 166
dj_ic_lmloc (fortran variable in module fields), 165
dj_lmloc (fortran variable in module fields), 165
dj_rloc (fortran variable in module fields), 165
djdt_iclast (fortran variable in module fieldslast), 168
djdt_iclast_lmloc (fortran variable in module fieldslast), 168
djdt_lmloc (fortran variable in module step_time_mod), 203
djdt_rloc (fortran variable in module step_time_mod), 202
djdtlast (fortran variable in module fieldslast), 168
djdtlast_lmloc (fortran variable in module fieldslast), 168
dlbmean (fortran variable in module output_mod), 275
dlh (fortran variable in module horizontal_data), 199
dlkappa (fortran variable in module radial_functions), 196
dllambda (fortran variable in module radial_functions), 196
dlvcmean (fortran variable in module output_mod), 275
dlvcmeanr (fortran variable in module outpar_mod), 287
dlvisc (fortran variable in module radial_functions), 196
dlvmean (fortran variable in module output_mod), 274
dlvmeanr (fortran variable in module outpar_mod), 288
dlvu2cmeanr (fortran variable in module outpar_mod), 288
dlvu2meanr (fortran variable in module outpar_mod), 288
dmbmean (fortran variable in module output_mod), 275

dmvmean (fortran variable in module output_mod), 275
do_iteration_thetablocking_openmp() (fortran subroutine in module riterthetablocking_openmp_mod), 227
dp (fortran variable in module fields), 166
dp_lmloc (fortran variable in module fields), 165
dp_rloc (fortran variable in module fields), 166
dpdt_lmloc (fortran variable in module step_time_mod), 203
dpdt_rloc (fortran variable in module step_time_mod), 202
dpdtlast (fortran variable in module fieldslast), 168
dpdtlast_lmloc (fortran variable in module fieldslast), 168
dphi (fortran variable in module horizontal_data), 199
dphi0 (fortran variable in module horizontal_data), 199
dphi02 (fortran variable in module horizontal_data), 199
dpl0eq (fortran variable in module horizontal_data), 200
dplm (fortran variable in module horizontal_data), 199
dplms (fortran variable in module egeos_mod), 348
dplms (fortran variable in module outpv3), 350
dplms (fortran variable in module outto_mod), 344
dplmz (fortran variable in module outpv3), 350
dpvmean (fortran variable in module output_mod), 275
dr_fac (fortran variable in module radial_functions), 196
dr_fac_ic (fortran variable in module radial_functions), 196
dr_facc (fortran variable in module rms), 322
drx (fortran variable in module radial_functions), 198
ds (fortran variable in module fields), 166
ds_lmloc (fortran variable in module fields), 166
ds_rloc (fortran variable in module fields), 167
dsdt_lmloc (fortran variable in module step_time_mod), 202
dsdt_rloc (fortran variable in module step_time_mod), 202
dsdtlast (fortran variable in module fieldslast), 168
dsdtlast_lmloc (fortran variable in module fieldslast), 168
dt_bpot (fortran variable in module output_data), 175
dt_cmb (fortran variable in module output_data), 172
dt_cmb_file (fortran variable in module output_data), 171
dt_courant() (fortran subroutine in module courant_mod), 204
dt_graph (fortran variable in module output_data), 171
dt_icb2_ave (fortran variable in module spectra), 293
dt_icb_ave (fortran variable in module spectra), 293
dt_log (fortran variable in module output_data), 172
dt_movie (fortran variable in module output_data), 172
dt_pot (fortran variable in module output_data), 173
dt_r_field (fortran variable in module output_data), 172
dt_rst (fortran variable in module output_data), 174
dt_spec (fortran variable in module output_data), 171
dt_to (fortran variable in module output_data), 174
dt_tomovie (fortran variable in module output_data), 171
dt_toz (fortran variable in module output_data), 172
dt_tpot (fortran variable in module output_data), 171
dt_vpot (fortran variable in module output_data), 172
dtb_arrays_mod (module), 333
dtb_gather_rloc_on_rank0() (fortran subroutine in module dtb_mod), 331
dtb_mod (module), 328
dtbpol2hint (fortran variable in module rms), 321
dtbpollmr (fortran variable in module rms), 322
dtbrms() (fortran subroutine in module rms), 324
dtbrms_file (fortran variable in module output_data), 174
dtbtor2hint (fortran variable in module rms), 322
dteint (fortran variable in module output_mod), 275
dtemp0 (fortran variable in module radial_functions), 196
dtheta1a (fortran variable in module horizontal_data), 200
dtheta1s (fortran variable in module horizontal_data), 200
dtheta2a (fortran variable in module horizontal_data), 200
dtheta2s (fortran variable in module horizontal_data), 200
dtheta3a (fortran variable in module horizontal_data), 200
dtheta3s (fortran variable in module horizontal_data), 200
dtheta4a (fortran variable in module horizontal_data), 199
dtheta4s (fortran variable in module horizontal_data), 200
dtmax (fortran variable in module num_param), 157
dtmin (fortran variable in module num_param), 156
dtp (fortran variable in module updateb_mod), 219
dtstart (fortran variable in module num_param), 156
dtt (fortran variable in module updateb_mod), 219
dtv (fortran variable in module updatewp_mod), 211
dtv (fortran variable in module updatez_mod), 213
dtvpol2hint (fortran variable in module rms), 322
dtvpollmr (fortran variable in module rms), 322
dtvrms() (fortran subroutine in module rms), 324
dtvrms_file (fortran variable in module output_data), 173
dtvrmsl_sd (fortran variable in module rms), 322
dtvrmsl_ta (fortran variable in module rms), 322
dtvrmssd (fortran variable in module rms), 321
dtvtor2hint (fortran variable in module rms), 321
duhmeanr (fortran variable in module outpar_mod), 288
dvpm (fortran variable in module outto_mod), 344
dvsrlm_lmloc (fortran variable in module step_time_mod), 202
dvsrlm_rloc (fortran variable in module step_time_mod), 203
dvxbhlm_lmloc (fortran variable in module step_time_mod), 203
dvxbhlm_rloc (fortran variable in module step_time_mod), 202
dw (fortran variable in module fields), 166
dw_ave_global (fortran variable in module fields_average_mod), 319
dw_lmloc (fortran variable in module fields), 165
dw_rloc (fortran variable in module fields), 166
dwdt_lmloc (fortran variable in module step_time_mod), 203

dwdt_rloc (fortran variable in module step_time_mod), 202
dwdtlast (fortran variable in module fieldslast), 168
dwdtlast_lmloc (fortran variable in module fieldslast), 168
dz (fortran variable in module fields), 165
dz_lmloc (fortran variable in module fields), 167
dz_rloc (fortran variable in module fields), 166
dzastrlmr (fortran variable in module torsional_oscillations), 338
dzastrlmr_rloc (fortran variable in module torsional_oscillations), 337
dzcorlmr (fortran variable in module torsional_oscillations), 338
dzcorlmr_rloc (fortran variable in module torsional_oscillations), 338
dzddvplmr (fortran variable in module torsional_oscillations), 338
dzddvplmr_rloc (fortran variable in module torsional_oscillations), 338
dzdt_lmloc (fortran variable in module step_time_mod), 202
dzdt_rloc (fortran variable in module step_time_mod), 203
dzdtlast (fortran variable in module fieldslast), 168
dzdtlast_lo (fortran variable in module fieldslast), 168
dzdvplmr (fortran variable in module torsional_oscillations), 337
dzdvplmr_rloc (fortran variable in module torsional_oscillations), 338
dzlflmr (fortran variable in module torsional_oscillations), 338
dzlflmr_rloc (fortran variable in module torsional_oscillations), 338
dzrstrlmr (fortran variable in module torsional_oscillations), 338
dzrstrlmr_rloc (fortran variable in module torsional_oscillations), 337
dzstrlmr (fortran variable in module torsional_oscillations), 338
dzstrlmr_rloc (fortran variable in module torsional_oscillations), 338
dzvmean (fortran variable in module output_mod), 275

E

e_cmb2_l_ave (fortran variable in module spectra), 292
e_cmb2_m_ave (fortran variable in module spectra), 292
e_cmb_l_ave (fortran variable in module spectra), 292
e_cmb_m_ave (fortran variable in module spectra), 293
e_dipa (fortran variable in module magnetic_energy), 280
e_kin_file (fortran variable in module output_data), 174
e_kin_pmean (fortran variable in module output_mod), 275
e_kin_tmean (fortran variable in module output_mod), 276
e_mag_ic_file (fortran variable in module output_data), 172
e_mag_oc_file (fortran variable in module output_data), 174
e_mag_pmean (fortran variable in module output_mod), 275
e_mag_tmean (fortran variable in module output_mod), 275
e_p2_l_ave (fortran variable in module spectra), 292
e_p2_m_ave (fortran variable in module spectra), 293
e_p_asa (fortran variable in module kinetic_energy), 278
e_p_asa (fortran variable in module magnetic_energy), 280
e_p_l_ave (fortran variable in module spectra), 292
e_p_m_ave (fortran variable in module spectra), 292
e_pa (fortran variable in module kinetic_energy), 278
e_pa (fortran variable in module magnetic_energy), 280
e_t2_l_ave (fortran variable in module spectra), 292
e_t2_m_ave (fortran variable in module spectra), 293
e_t_asa (fortran variable in module kinetic_energy), 278
e_t_asa (fortran variable in module magnetic_energy), 280
e_t_l_ave (fortran variable in module spectra), 293
e_t_m_ave (fortran variable in module spectra), 293
e_ta (fortran variable in module kinetic_energy), 278
e_ta (fortran variable in module magnetic_energy), 280
egeos_mod (module), 347
ek (fortran variable in module physical_parameters), 158
ek_p2_l_ave (fortran variable in module spectra), 293
ek_p2_m_ave (fortran variable in module spectra), 293
ek_p_l_ave (fortran variable in module spectra), 292
ek_p_m_ave (fortran variable in module spectra), 292
ek_t2_l_ave (fortran variable in module spectra), 293
ek_t2_m_ave (fortran variable in module spectra), 293
ek_t_l_ave (fortran variable in module spectra), 292
ek_t_m_ave (fortran variable in module spectra), 293
ekscaled (fortran variable in module physical_parameters), 160
elcmbmean (fortran variable in module output_mod), 275
elmean (fortran variable in module output_mod), 275
enscale (fortran variable in module num_param), 157
eparaximeanr (fortran variable in module outpar_mod), 288
eparmeanr (fortran variable in module outpar_mod), 287
eperpaximeanr (fortran variable in module outpar_mod), 288
eperpmeanr (fortran variable in module outpar_mod), 288
eps (fortran variable in module egeos_mod), 347
eps (fortran variable in module rms), 323
epsc (fortran variable in module physical_parameters), 159
epsc0 (fortran variable in module physical_parameters), 158

epscprof (fortran variable in module radial_functions), **196**
epss (fortran variable in module physical_parameters), **160**
equat() (magic.cyl.Cyl method), 140
equat() (magic.Surf method), 125
escale (fortran variable in module num_param), **157**
etot (fortran variable in module output_mod), **274**
etotold (fortran variable in module output_mod), **275**
expo_imp (fortran variable in module bext), **176**
ExtraPot (class in magic.potExtra), 135

F

fac_loop (fortran variable in module bext), **176**
factorise() (fortran subroutine in module useful), **365**
fast_read() (in module magic.libmagic), 145
fcondmeanr (fortran variable in module outpar_mod), **288**
fconvmeanr (fortran variable in module outpar_mod), **287**
fft (module), **252**
fft99ajw() (fortran subroutine in module fft), **255**
fft99bjw() (fortran subroutine in module fft), **255**
fft_fac_complex() (fortran subroutine in module fft_fac_mod), **241**
fft_fac_mod (module), **240**
fft_fac_real() (fortran subroutine in module fft_fac_mod), **240**
fft_thetab() (fortran subroutine in module fft), **253**
fft_to_real() (fortran subroutine in module fft), **253**
fftjw() (fortran subroutine in module fft), **254**
fields (module), **164**
fields_average() (fortran subroutine in module fields_average_mod), **319**
fields_average_mod (module), **317**
fieldslast (module), **167**
final_wait_array (fortran variable in module communications), **183**
finalize() (fortran subroutine in module cosine_transform_even), **239**
finalize() (fortran subroutine in module cosine_transform_odd), **237**
finalize() (fortran subroutine in module dtb_arrays_mod), **333**
finalize() (fortran subroutine in module nonlinear_lm_mod), **231**
finalize() (fortran subroutine in module to_arrays_mod), **343**
finalize_movie_data() (fortran subroutine in module movie_data), **300**
finalize_radialloop() (fortran subroutine in module radialloop), **221**
finalize_riterthetablocking_openmp() (fortran subroutine in module riterthetablocking_openmp_mod), **227**
finalize_updatewp() (fortran subroutine in module updatewp_mod), **211**
fkinmeanr (fortran variable in module outpar_mod), **288**
four (fortran variable in module constants), **177**
fourier2D() (magic.Butterfly method), 143
fpoynmeanr (fortran variable in module outpar_mod), **288**
frames (fortran variable in module movie_data), **299**
fresmeanr (fortran variable in module outpar_mod), **288**
fviscmeanr (fortran variable in module outpar_mod), **288**

G

g0 (fortran variable in module physical_parameters), **159**
g1 (fortran variable in module physical_parameters), **159**
g2 (fortran variable in module physical_parameters), **159**
gather_all_from_lo_to_rank0() (fortran subroutine in module communications), **184**
gather_from_lo_to_rank0() (fortran subroutine in module communications), **184**
gauleg() (fortran subroutine in module horizontal_data), **200**
gauss (fortran variable in module horizontal_data), **200**
general_arrays_mod (module), **230**
geo2hint (fortran variable in module rms), **321**
geormsl_sd (fortran variable in module rms), **322**
geormsl_ta (fortran variable in module rms), **322**
geormssd (fortran variable in module rms), **323**
geosmean (fortran variable in module output_mod), **275**
get_angular_moment() (fortran subroutine in module outrot), **286**
get_b_nl_bcs() (fortran subroutine in module nonlinear_bcs), **233**
get_b_surface() (fortran subroutine in module out_movie), **312**
get_bmat() (fortran subroutine in module updateb_mod), **220**
get_bpol() (fortran subroutine in module out_dtb_frame), **335**
get_br_v_bcs() (fortran subroutine in module nonlinear_bcs), **233**
get_btor() (fortran subroutine in module out_dtb_frame), **335**
get_chebs (fortran variable in module chebyshev_polynoms_mod), **235**
get_chebs_direct() (fortran subroutine in module chebyshev_polynoms_mod), **236**
get_chebs_even() (fortran subroutine in module chebyshev_polynoms_mod), **235**
get_chebs_recurr() (fortran subroutine in module chebyshev_polynoms_mod), **235**
get_dcheb (fortran variable in module radial_der), **259**
get_dcheb_complex() (fortran subroutine in module radial_der), **262**
get_dcheb_even() (fortran subroutine in module radial_der_even), **265**

get_dcheb_real_1d() (fortran subroutine in module radial_der), 262
get_ddcheb() (fortran subroutine in module radial_der), 262
get_ddcheb_even() (fortran subroutine in module radial_der_even), 266
get_dddcheb() (fortran subroutine in module radial_der), 263
get_dddr() (fortran subroutine in module radial_der), 261
get_ddr() (fortran subroutine in module radial_der), 261
get_ddr_even() (fortran subroutine in module radial_der_even), 264
get_ddrns_even() (fortran subroutine in module radial_der_even), 265
get_dh_dtblm() (fortran subroutine in module dtb_mod), 332
get_dr (fortran variable in module radial_der), 259
get_dr_complex() (fortran subroutine in module radial_der), 259
get_dr_real_1d() (fortran subroutine in module radial_der), 260
get_drns() (fortran subroutine in module radial_der), 260
get_drns_even() (fortran subroutine in module radial_der_even), 264
get_dtb() (fortran subroutine in module out_dtb_frame), 335
get_dtblm() (fortran subroutine in module dtb_mod), 331
get_dtblmfinish() (fortran subroutine in module dtb_mod), 331
get_e_kin() (fortran subroutine in module kinetic_energy), 278
get_e_mag() (fortran subroutine in module magnetic_energy), 280
get_fl() (fortran subroutine in module out_movie), 312
get_fluxes() (fortran subroutine in module nl_special_calc), 356
get_global_sum (fortran variable in module communications), 182
get_global_sum_cmplx_1d() (fortran function in module communications), 183
get_global_sum_cmplx_2d() (fortran function in module communications), 183
get_global_sum_real_2d() (fortran function in module communications), 183
get_helicity() (fortran subroutine in module nl_special_calc), 357
get_hit_times() (fortran subroutine in module precalculations), 194
get_lorder_lm_blocking() (fortran subroutine in module blocking), 270
get_lorentz_torque() (fortran subroutine in module outrot), 286
get_movie_type() (fortran subroutine in module movie_data), 300
get_nlblayers() (fortran subroutine in module nl_special_calc), 355
get_p0mat() (fortran subroutine in module updatewp_mod), 212
get_pas() (fortran subroutine in module to_helpers), 341
get_paslm() (fortran subroutine in module rms_helpers), 325
get_perppar() (fortran subroutine in module nl_special_calc), 356
get_poltorrms() (fortran subroutine in module rms_helpers), 325
get_power() (fortran subroutine in module power), 290
get_ras() (fortran subroutine in module rms_helpers), 328
get_s0mat() (fortran subroutine in module updates_mod), 217
get_sl() (fortran subroutine in module out_movie), 311
get_smat() (fortran subroutine in module updates_mod), 217
get_snake_lm_blocking() (fortran subroutine in module blocking), 270
get_standard_deviation() (fortran function in module spectra), 293
get_standard_lm_blocking() (fortran subroutine in module blocking), 270
get_subblocks() (fortran subroutine in module blocking), 270
get_td() (fortran subroutine in module nonlinear_lm_mod), 231
get_theta_blocking (fortran variable in module blocking), 269
get_theta_blocking_cache() (fortran subroutine in module blocking), 270
get_theta_blocking_openmp() (fortran subroutine in module blocking), 270
get_u_square() (fortran subroutine in module kinetic_energy), 279
get_viscous_torque (fortran variable in module outrot), 285
get_viscous_torque_complex() (fortran subroutine in module outrot), 285
get_viscous_torque_real() (fortran subroutine in module outrot), 285
get_wpmat() (fortran subroutine in module updatewp_mod), 211
get_z10mat() (fortran subroutine in module updatez_mod), 214
get_zmat() (fortran subroutine in module updatez_mod), 214
getAccuratePeaks() (in module magic.bLayers), 137
getastr() (fortran subroutine in module to_helpers), 342
getbackground() (fortran subroutine in module radial_functions), 198
getCpuTime() (in module magic.libmagic), 145
getdlm() (fortran subroutine in module getdlm_mod), 282

getdlm_mod (module), 281
getdvptr() (fortran subroutine in module egeos_mod), 348
getegeos() (fortran subroutine in module egeos_mod), 348
getGauss() (in module magic.coeff), 131
getlm2lmo() (fortran subroutine in module readcheckpoints), 360
getMaxima() (in module magic.bLayers), 137
getmsd2() (fortran subroutine in module useful), 366
getpastr() (fortran subroutine in module to_helpers), 341
getpvptr() (fortran subroutine in module outpv3), 351
getstartfields() (fortran subroutine in module start_fields), 188
getthistype() (fortran function in module riterthetablocking_openmp_mod), 227
getto() (fortran subroutine in module torsional_oscillations), 338
gettofinish() (fortran subroutine in module torsional_oscillations), 340
gettonext() (fortran subroutine in module torsional_oscillations), 339
getTotalRunTime() (in module magic.libmagic), 145
gradt2meanr (fortran variable in module outpar_mod), 288
Graph2Vtk (class in magic.graph2vtk), 133
graph_file (fortran variable in module output_data), 175
graph_mpi_fh (fortran variable in module output_data), 173
graph_write() (fortran subroutine in module graphout_mod), 297
graph_write_mpi() (fortran subroutine in module graphout_mod), 298
graphout() (fortran subroutine in module graphout_mod), 296
graphout_header() (fortran subroutine in module graphout_mod), 296
graphout_ic() (fortran subroutine in module graphout_mod), 297
graphout_mod (module), 295
graphout_mpi() (fortran subroutine in module graphout_mod), 296
graphout_mpi_header() (fortran subroutine in module graphout_mod), 297
grenoble (module), 178
gt_cheb (fortran variable in module communications), 183
gt_ic (fortran variable in module communications), 182
gt_oc (fortran variable in module communications), 182

H

half (fortran variable in module constants), 178
hammer2cart() (in module magic.libmagic), 146
hdf5helpers (module), 368
hdif_b (fortran variable in module horizontal_data), 199
hdif_s (fortran variable in module horizontal_data), 200
hdif_v (fortran variable in module horizontal_data), 200
hint2dpol() (fortran subroutine in module rms_helpers), 326
hint2dpollm() (fortran subroutine in module rms_helpers), 326
hint2pol() (fortran subroutine in module rms_helpers), 326
hint2pollm() (fortran subroutine in module rms_helpers), 327
hint2tor() (fortran subroutine in module rms_helpers), 327
hint2torlm() (fortran subroutine in module rms_helpers), 328
hintrms() (fortran subroutine in module rms_helpers), 327
horizontal() (fortran subroutine in module horizontal_data), 200
horizontal_data (module), 198

I

i_fft_init (fortran variable in module fft), 253
ierr (fortran variable in module parallel_mod), 179
imagcon (fortran variable in module physical_parameters), 160
imps (fortran variable in module physical_parameters), 160
inform (fortran variable in module init_fields), 191
init_b1 (fortran variable in module init_fields), 190
init_fft() (fortran subroutine in module fft), 253
init_fields (module), 189
init_rnb() (fortran subroutine in module rms), 323
init_s1 (fortran variable in module init_fields), 190
init_s2 (fortran variable in module init_fields), 191
init_v1 (fortran variable in module init_fields), 190
initb() (fortran subroutine in module init_fields), 192
initialize() (fortran subroutine in module cosine_transform_even), 239
initialize() (fortran subroutine in module cosine_transform_odd), 237
initialize() (fortran subroutine in module dtb_arrays_mod), 333
initialize() (fortran subroutine in module leg_helper_mod), 244
initialize() (fortran subroutine in module nonlinear_lm_mod), 231
initialize() (fortran subroutine in module to_arrays_mod), 343
initialize_blocking() (fortran subroutine in module blocking), 270
initialize_communications() (fortran subroutine in module communications), 183
initialize_dtb_mod() (fortran subroutine in module dtb_mod), 331
initialize_egeos_mod() (fortran subroutine in module egeos_mod), 348

initialize_fields() (fortran subroutine in module fields), 167
initialize_fields_average_mod() (fortran subroutine in module fields_average_mod), 319
initialize_fieldslast() (fortran subroutine in module fieldslast), 168
initialize_grenoble() (fortran subroutine in module grenoble), 179
initialize_horizontal_data() (fortran subroutine in module horizontal_data), 200
initialize_init_fields() (fortran subroutine in module init_fields), 191
initialize_kinetic_energy() (fortran subroutine in module kinetic_energy), 278
initialize_lmloop() (fortran subroutine in module lmloop_mod), 208
initialize_lmloop_data() (fortran subroutine in module lmloop_data), 181
initialize_magnetic_energy() (fortran subroutine in module magnetic_energy), 280
initialize_matrices() (fortran subroutine in module matrices), 170
initialize_movie_data() (fortran subroutine in module movie_data), 300
initialize_num_param() (fortran subroutine in module num_param), 157
initialize_outpar_mod() (fortran subroutine in module outpar_mod), 288
initialize_output() (fortran subroutine in module output_mod), 276
initialize_output_power() (fortran subroutine in module power), 290
initialize_outpv3() (fortran subroutine in module outpv3), 350
initialize_outto_mod() (fortran subroutine in module outto_mod), 344
initialize_radial_data() (fortran subroutine in module radial_data), 180
initialize_radial_functions() (fortran subroutine in module radial_functions), 198
initialize_radialloop() (fortran subroutine in module radialloop), 221
initialize_riterthetablocking_openmp() (fortran subroutine in module riterthetablocking_openmp_mod), 227
initialize_rms() (fortran subroutine in module rms), 323
initialize_spectra() (fortran subroutine in module spectra), 293
initialize_step_time() (fortran subroutine in module step_time_mod), 203
initialize_to() (fortran subroutine in module torsional_oscillations), 338
initialize_truncation() (fortran subroutine in module truncation), 155
initialize_updateb() (fortran subroutine in module updateb_mod), 219
initialize_updates() (fortran subroutine in module updates_mod), 216
initialize_updatewp() (fortran subroutine in module updatewp_mod), 211
initialize_updatez() (fortran subroutine in module updatez_mod), 213
inits() (fortran subroutine in module init_fields), 191
initv() (fortran subroutine in module init_fields), 191
intcheb() (in module magic.libmagic), 146
integBotTop() (in module magic.bLayers), 137
integBulkBc() (in module magic.bLayers), 137
integration (module), 266
interior_model (fortran variable in module physical_parameters), 160
intfac (fortran variable in module num_param), 157
istop (fortran variable in module num_param), 156

J

j_cond() (fortran subroutine in module init_fields), 192
jmat (fortran variable in module matrices), 169
jmat_fac (fortran variable in module matrices), 169
jpivot (fortran variable in module matrices), 169
jvarcon (fortran variable in module radial_functions), 196

K

kappa (fortran variable in module radial_functions), 196
kbotb (fortran variable in module physical_parameters), 160
kbots (fortran variable in module physical_parameters), 159
kbotv (fortran variable in module physical_parameters), 160
kinetic_energy (module), 277
ktopb (fortran variable in module physical_parameters), 158
ktops (fortran variable in module physical_parameters), 158
ktopv (fortran variable in module physical_parameters), 158

L

l2lmas (fortran variable in module blocking), 269
l_am (fortran variable in module logic), 162
l_anel (fortran variable in module logic), 163
l_anelastic_liquid (fortran variable in module logic), 163
l_average (fortran variable in module logic), 162
l_b_nl_cmb (fortran variable in module logic), 164
l_b_nl_icb (fortran variable in module logic), 163
l_cmb_field (fortran variable in module logic), 163
l_cond_ic (fortran variable in module logic), 162
l_cond_ma (fortran variable in module logic), 163
l_conv (fortran variable in module logic), 162

l_conv_nl (fortran variable in module logic), 163
l_corr (fortran variable in module logic), 164
l_correct_ame (fortran variable in module logic), 161
l_correct_amz (fortran variable in module logic), 161
l_correct_step() (fortran function in module useful), 364
l_corrmov (fortran variable in module logic), 163
l_curr (fortran variable in module bext), 176
l_drift (fortran variable in module logic), 163
l_dt_cmb_field (fortran variable in module logic), 163
l_dtb (fortran variable in module logic), 164
l_dtbmovie (fortran variable in module logic), 161
l_dtrmagspec (fortran variable in module logic), 161
l_fluxprofs (fortran variable in module logic), 163
l_graph_time (fortran variable in module output_data), 172
l_heat (fortran variable in module logic), 161
l_heat_nl (fortran variable in module logic), 164
l_hel (fortran variable in module logic), 163
l_ht (fortran variable in module logic), 164
l_htmovie (fortran variable in module logic), 163
l_imp (fortran variable in module bext), 176
l_iner (fortran variable in module logic), 164
l_isothermal (fortran variable in module logic), 162
l_lcr (fortran variable in module logic), 161
l_mag (fortran variable in module logic), 162
l_mag_kin (fortran variable in module logic), 164
l_mag_lf (fortran variable in module logic), 161
l_mag_nl (fortran variable in module logic), 162
l_max (fortran variable in module truncation), 153
l_max_cmb (fortran variable in module output_data), 172
l_max_r (fortran variable in module output_data), 174
l_maxmag (fortran variable in module truncation), 155
l_movie (fortran variable in module logic), 163
l_movie_ic (fortran variable in module logic), 162
l_movie_oc (fortran variable in module logic), 163
l_newmap (fortran variable in module logic), 163
l_non_rot (fortran variable in module logic), 162
l_par (fortran variable in module logic), 164
l_perppar (fortran variable in module logic), 162
l_power (fortran variable in module logic), 164
l_pressgraph (fortran variable in module logic), 162
l_pv (fortran variable in module logic), 161
l_r_field (fortran variable in module logic), 164
l_r_fieldt (fortran variable in module logic), 162
l_reset_t (fortran variable in module init_fields), 191
l_rmagspec (fortran variable in module logic), 163
l_rms (fortran variable in module logic), 164
l_rot_ic (fortran variable in module logic), 163
l_rot_ma (fortran variable in module logic), 164
l_runtimelimit (fortran variable in module logic), 163
l_save_out (fortran variable in module logic), 163
l_sric (fortran variable in module logic), 162
l_srma (fortran variable in module logic), 161
l_start_file (fortran variable in module init_fields), 190
l_store_frame (fortran variable in module logic), 162
l_storebpot (fortran variable in module logic), 162
l_storepot (fortran variable in module logic), 164
l_storetpot (fortran variable in module logic), 163
l_storevpot (fortran variable in module logic), 162
l_time_hits (fortran variable in module logic), 161
l_to (fortran variable in module logic), 161
l_tomovie (fortran variable in module logic), 164
l_true_time (fortran variable in module logic), 162
l_update_b (fortran variable in module logic), 162
l_update_s (fortran variable in module logic), 162
l_update_v (fortran variable in module logic), 162
l_viscbccalc (fortran variable in module logic), 162
l_z10mat (fortran variable in module logic), 163
lambda (fortran variable in module radial_functions), 198
lavemem (fortran variable in module truncation), 154
lbdissmean (fortran variable in module output_mod), 276
lbmat (fortran variable in module matrices), 169
ldif (fortran variable in module num_param), 156
ldifexp (fortran variable in module num_param), 156
ldtbmem (fortran variable in module truncation), 154
leg_helper_mod (module), 242
legendre_grid_to_spec (module), 250
legendre_spec_to_grid (module), 246
legprep() (fortran subroutine in module leg_helper_mod), 244
legprep_ic() (fortran subroutine in module leg_helper_mod), 245
legprepg() (fortran subroutine in module leg_helper_mod), 244
legtf() (fortran subroutine in module legendre_spec_to_grid), 249
legtf1() (fortran subroutine in module legendre_grid_to_spec), 251
legtf2() (fortran subroutine in module legendre_grid_to_spec), 251
legtf3() (fortran subroutine in module legendre_grid_to_spec), 251
legtfas() (fortran subroutine in module legendre_grid_to_spec), 251
legtfas2() (fortran subroutine in module legendre_grid_to_spec), 252
legtfg() (fortran subroutine in module legendre_spec_to_grid), 246
legtfgnomag() (fortran subroutine in module legendre_spec_to_grid), 248
length_to_blank() (fortran function in module charmanip), 367
length_to_char() (fortran function in module charmanip), 367
lf2hint (fortran variable in module rms), 322
lffac (fortran variable in module physical_parameters), 158
lfm (fortran variable in module outto_mod), 344

lfrmsl_sd (fortran variable in module rms), 322
lfrmsl_ta (fortran variable in module rms), 323
lfrmssd (fortran variable in module rms), 321
lgeos (fortran variable in module truncation), 153
lgrenoble (fortran variable in module grenoble), 178
licfield (fortran variable in module movie_data), 299
lip (fortran variable in module precision_mod), 153
llm (fortran variable in module lmloop_data), 181
llmmag (fortran variable in module lmloop_data), 181
lm2 (fortran variable in module blocking), 268
lm22l (fortran variable in module blocking), 269
lm22lm (fortran variable in module blocking), 269
lm22m (fortran variable in module blocking), 269
lm2l (fortran variable in module blocking), 269
lm2lma (fortran variable in module blocking), 269
lm2lmp (fortran variable in module blocking), 269
lm2lms (fortran variable in module blocking), 269
lm2lo_redist() (fortran subroutine in module communications), 185
lm2m (fortran variable in module blocking), 269
lm2mc (fortran variable in module blocking), 269
lm2pt() (fortran subroutine in module out_dtb_frame), 336
lm2r_redist_start() (fortran subroutine in module communications), 185
lm2r_redist_wait() (fortran subroutine in module communications), 185
lm_max (fortran variable in module truncation), 154
lm_max_ave (fortran variable in module truncation), 154
lm_max_dtb (fortran variable in module truncation), 155
lm_max_real (fortran variable in module truncation), 155
lm_maxgeos (fortran variable in module truncation), 154
lm_maxmag (fortran variable in module truncation), 154
lm_on_last_rank (fortran variable in module lmloop_data), 181
lm_per_rank (fortran variable in module lmloop_data), 181
lmagmem (fortran variable in module truncation), 155
lmas2pt() (fortran subroutine in module legendre_spec_to_grid), 250
lmloop() (fortran subroutine in module lmloop_mod), 209
lmloop_data (module), 180
lmloop_mod (module), 208
lmmapping (module), 271
lmmaxs (fortran variable in module outto_mod), 344
lmodd (fortran variable in module horizontal_data), 199
lmoddp (fortran variable in module horizontal_data), 200
lmoviemem (fortran variable in module truncation), 154
lmp2 (fortran variable in module blocking), 268
lmp2l (fortran variable in module blocking), 269
lmp2lm (fortran variable in module blocking), 269
lmp2lmpa (fortran variable in module blocking), 269
lmp2lmps (fortran variable in module blocking), 269
lmp_max (fortran variable in module truncation), 154
lmp_max_dtb (fortran variable in module truncation), 154
lmstartb (fortran variable in module blocking), 268
lmstopb (fortran variable in module blocking), 269
lnegtime() (fortran function in module timing), 207
lnpas2tr() (fortran function in module omega), 354
lo2lm_redist() (fortran subroutine in module communications), 186
lo2r_aj (fortran variable in module communications), 183
lo2r_b (fortran variable in module communications), 183
lo2r_p (fortran variable in module communications), 183
lo2r_redist_start() (fortran subroutine in module communications), 185
lo2r_redist_wait() (fortran subroutine in module communications), 185
lo2r_s (fortran variable in module communications), 183
lo2r_w (fortran variable in module communications), 183
lo2r_z (fortran variable in module communications), 183
lo_map (fortran variable in module blocking), 269
lo_sub_map (fortran variable in module blocking), 269
log_file (fortran variable in module output_data), 175
logic (module), 161
logwrite() (fortran subroutine in module useful), 366
lorentz_torque_iclast (fortran variable in module field_slast), 168
lorentz_torque_malast (fortran variable in module field_slast), 168
lp_file (fortran variable in module output_data), 175
lscale (fortran variable in module num_param), 157
lsmat (fortran variable in module matrices), 169
lstart (fortran variable in module horizontal_data), 200
lstartp (fortran variable in module horizontal_data), 200
lstop (fortran variable in module horizontal_data), 199
lstopp (fortran variable in module horizontal_data), 199
lstoremov (fortran variable in module movie_data), 299
lstressmem (fortran variable in module truncation), 153
ltimelimit() (fortran function in module timing), 206
lvdissmean (fortran variable in module output_mod), 275
lverbose (fortran variable in module logic), 161
lwpmat (fortran variable in module matrices), 169
lz10mat (fortran variable in module matrices), 169
lzmat (fortran variable in module matrices), 169

M

m_max (fortran variable in module truncation), 154
mag2hint (fortran variable in module rms), 323
magic (fortran program), 151
magic.bLayers (module), 136
magic.coeff (module), 130
magic.cyl (module), 138
magic.graph2vtk (module), 133
magic.libmagic (module), 143
magic.potExtra (module), 135
MagicCoeffCmb (class in magic.coeff), 130
MagicCoeffR (class in magic.coeff), 130

MagicGraph (class in magic), 123
MagicRadial (class in magic), 121
MagicRSpec (class in magic), 131
MagicSetup (class in magic), 119
MagicSpectrum (class in magic), 122
MagicSpectrum2D (class in magic), 122
MagicTs (class in magic), 120
magnetic_energy (module), **279**
magrmsl_sd (fortran variable in module rms), **322**
magrmsl_ta (fortran variable in module rms), **323**
magrmssd (fortran variable in module rms), **323**
mapdatahydro() (fortran subroutine in module readcheckpoints), **360**
mapdatamag() (fortran subroutine in module readcheckpoints), **361**
mapdatar() (fortran subroutine in module readcheckpoints), **362**
mass (fortran variable in module constants), **178**
matder() (in module magic.libmagic), 146
matrices (module), **168**
maxthreads (fortran variable in module updateb_mod), **218**
maxthreads (fortran variable in module updates_mod), **216**
maxthreads (fortran variable in module updatewp_mod), **210**
maxthreads (fortran variable in module updatez_mod), **213**
meantime() (fortran subroutine in module timing), **207**
minc (fortran variable in module truncation), **153**
misc_file (fortran variable in module output_data), **174**
mode (fortran variable in module physical_parameters), **159**
Movie (class in magic), 127
movie (fortran variable in module movie_data), **299**
Movie3D (class in magic), 129
movie_const (fortran variable in module movie_data), **299**
movie_data (module), **298**
movie_file (fortran variable in module movie_data), **299**
movie_gather_frames_to_rank0() (fortran subroutine in module movie_data), **306**
moviedipcolat (fortran variable in module movie_data), **299**
moviediplon (fortran variable in module movie_data), **299**
moviedipstrength (fortran variable in module movie_data), **299**
moviedipstrengthgeo (fortran variable in module movie_data), **299**
mpi_def_complex (fortran variable in module precision_mod), **153**
mpi_def_real (fortran variable in module precision_mod), **153**
mpi_out_real (fortran variable in module precision_mod), **152**
ms2time() (fortran subroutine in module timing), **206**
msechour (fortran variable in module timing), **205**
msecminute (fortran variable in module timing), **206**
msecsecond (fortran variable in module timing), **205**
mvar (fortran variable in module outpar_mod), **288**
myallgather() (fortran subroutine in module communications), **186**

N

n_angular_file (fortran variable in module output_data), **175**
n_b_r_file (fortran variable in module output_data), **175**
n_b_r_sets (fortran variable in module output_mod), **275**
n_bpot_step (fortran variable in module output_data), **175**
n_bpots (fortran variable in module output_data), **172**
n_cheb_ic_max (fortran variable in module truncation), **153**
n_cheb_max (fortran variable in module truncation), **155**
n_cheb_maxc (fortran variable in module rms), **322**
n_cmb_file (fortran variable in module output_data), **175**
n_cmb_setsmov (fortran variable in module output_mod), **275**
n_cmb_step (fortran variable in module output_data), **172**
n_cmbmov_file (fortran variable in module output_data), **174**
n_cmbs (fortran variable in module output_data), **174**
n_coeff_r (fortran variable in module output_data), **174**
n_coeff_r_max (fortran variable in module output_data), **175**
n_cour_step (fortran variable in module num_param), **157**
n_dipole_file (fortran variable in module output_data), **175**
n_dt_cmb_file (fortran variable in module output_data), **172**
n_dt_cmb_sets (fortran variable in module output_mod), **275**
n_dtbrms_file (fortran variable in module output_data), **173**
n_dtvrms_file (fortran variable in module output_data), **172**
n_e_kin_file (fortran variable in module output_data), **175**
n_e_mag_ic_file (fortran variable in module output_data), **175**
n_e_mag_oc_file (fortran variable in module output_data), **174**
n_e_sets (fortran variable in module output_mod), **275**
n_frame_work (fortran variable in module movie_data), **299**
n_graph_file (fortran variable in module output_data), **173**
n_graph_step (fortran variable in module output_data), **172**
n_graphs (fortran variable in module output_data), **171**

n_imp (fortran variable in module bext), 176
n_imps (fortran variable in module physical_parameters), 158
n_imps_max (fortran variable in module physical_parameters), 160
n_kin_spec_file (fortran variable in module output_data), 173
n_log_file (fortran variable in module output_data), 173
n_log_step (fortran variable in module output_data), 172
n_logs (fortran variable in module output_data), 174
n_lp_file (fortran variable in module output_data), 175
n_lscale (fortran variable in module num_param), 156
n_m_max (fortran variable in module truncation), 154
n_mag_spec_file (fortran variable in module output_data), 175
n_md (fortran variable in module movie_data), 299
n_misc_file (fortran variable in module output_data), 173
n_movie_const (fortran variable in module movie_data), 299
n_movie_field_start (fortran variable in module movie_data), 299
n_movie_field_stop (fortran variable in module movie_data), 299
n_movie_field_type (fortran variable in module movie_data), 299
n_movie_fields (fortran variable in module movie_data), 299
n_movie_fields_ic (fortran variable in module movie_data), 299
n_movie_fields_max (fortran variable in module movie_data), 299
n_movie_file (fortran variable in module movie_data), 299
n_movie_frames (fortran variable in module output_data), 171
n_movie_step (fortran variable in module output_data), 174
n_movie_surface (fortran variable in module movie_data), 299
n_movie_type (fortran variable in module movie_data), 299
n_movies (fortran variable in module movie_data), 299
n_movies_max (fortran variable in module movie_data), 299
n_par_file (fortran variable in module output_data), 171
n_perppar_file (fortran variable in module output_data), 173
n_phi_max (fortran variable in module truncation), 155
n_phi_maxstr (fortran variable in module truncation), 154
n_phi_tot (fortran variable in module truncation), 155
n_pot_step (fortran variable in module output_data), 174
n_pots (fortran variable in module output_data), 173
n_power_file (fortran variable in module output_data), 172
n_procs (fortran variable in module parallel_mod), 179
n_r_array (fortran variable in module output_data), 172
n_r_cmb (fortran variable in module radial_data), 180
n_r_field_step (fortran variable in module output_data), 175
n_r_fields (fortran variable in module output_data), 174
n_r_ic_max (fortran variable in module truncation), 153
n_r_ic_max_ave (fortran variable in module truncation), 155
n_r_ic_max_dtb (fortran variable in module truncation), 155
n_r_ic_maxmag (fortran variable in module truncation), 155
n_r_icb (fortran variable in module radial_data), 180
n_r_lcr (fortran variable in module physical_parameters), 159
n_r_max (fortran variable in module truncation), 154
n_r_max_ave (fortran variable in module truncation), 154
n_r_max_dtb (fortran variable in module truncation), 155
n_r_maxc (fortran variable in module rms), 323
n_r_maxgeos (fortran variable in module truncation), 154
n_r_maxmag (fortran variable in module truncation), 153
n_r_maxstr (fortran variable in module truncation), 155
n_r_step (fortran variable in module output_data), 171
n_r_tot (fortran variable in module truncation), 154
n_r_totmag (fortran variable in module truncation), 155
n_rot_file (fortran variable in module output_data), 173
n_rst_file (fortran variable in module output_data), 171
n_rst_step (fortran variable in module output_data), 173
n_rsts (fortran variable in module output_data), 173
n_s_bounds (fortran variable in module init_fields), 190
n_signal_file (fortran variable in module output_data), 174
n_spec (fortran variable in module output_mod), 275
n_spec_step (fortran variable in module output_data), 175
n_specs (fortran variable in module output_data), 173
n_sric_file (fortran variable in module output_data), 173
n_srma_file (fortran variable in module output_data), 174
n_start_file (fortran variable in module init_fields), 190
n_stores (fortran variable in module output_data), 174
n_t_bpot (fortran variable in module output_data), 172
n_t_cmb (fortran variable in module output_data), 171
n_t_graph (fortran variable in module output_data), 174
n_t_log (fortran variable in module output_data), 172
n_t_movie (fortran variable in module output_data), 173
n_t_pot (fortran variable in module output_data), 172
n_t_r_field (fortran variable in module output_data), 174
n_t_r_file (fortran variable in module output_data), 172
n_t_r_sets (fortran variable in module output_mod), 275
n_t_rst (fortran variable in module output_data), 175
n_t_spec (fortran variable in module output_data), 171
n_t_to (fortran variable in module output_data), 173
n_t_tomovie (fortran variable in module output_data), 172

n_t_toz (fortran variable in module output_data), 171
n_t_tpot (fortran variable in module output_data), 174
n_t_vpot (fortran variable in module output_data), 174
n_theta_cal2ord (fortran variable in module horizontal_data), 200
n_theta_max (fortran variable in module truncation), 154
n_theta_maxstr (fortran variable in module truncation), 154
n_time_hits (fortran variable in module output_data), 171
n_time_steps (fortran variable in module num_param), 156
n_to_step (fortran variable in module output_data), 172
n_tomovie_frames (fortran variable in module output_data), 173
n_tomovie_step (fortran variable in module output_data), 173
n_tos (fortran variable in module output_data), 173
n_toz_step (fortran variable in module output_data), 175
n_tozs (fortran variable in module output_data), 171
n_tpot_step (fortran variable in module output_data), 171
n_tpots (fortran variable in module output_data), 171
n_tscale (fortran variable in module num_param), 157
n_u2_spec_file (fortran variable in module output_data), 174
n_u_square_file (fortran variable in module output_data), 172
n_v_r_file (fortran variable in module output_data), 174
n_v_r_sets (fortran variable in module output_mod), 275
n_vpot_step (fortran variable in module output_data), 173
n_vpots (fortran variable in module output_data), 173
nalias (fortran variable in module truncation), 154
namelists (module), 186
nbdown (fortran variable in module blocking), 269
nbpotsets (fortran variable in module output_mod), 275
nbsave (fortran variable in module blocking), 268
ncp (fortran variable in module truncation), 154
ncpgeos (fortran variable in module truncation), 155
ncut (fortran variable in module rms), 321
nd (fortran variable in module fft), 253
ndd_costf1 (fortran variable in module radial_functions), 197
ndd_costf1_ic (fortran variable in module radial_functions), 198
ndd_costf2_ic (fortran variable in module radial_functions), 196
ndi_costf1 (fortran variable in module radial_functions), 197
ndi_costf1_ic (fortran variable in module radial_functions), 197
ndi_costf2_ic (fortran variable in module radial_functions), 196
nfs (fortran variable in module blocking), 269
ni (fortran variable in module fft), 253
nl_special_calc (module), 355
nlf (fortran variable in module output_data), 171
nlmbs (fortran variable in module blocking), 269
nlmbs2 (fortran variable in module blocking), 269
nlmbs_per_rank (fortran variable in module parallel_mod), 179
nlogs (fortran variable in module output_mod), 275
nonlinear_bcs (module), 232
nonlinear_lm_mod (module), 230
npvsets (fortran variable in module output_mod), 275
nr_on_last_rank (fortran variable in module parallel_mod), 179
nr_per_rank (fortran variable in module parallel_mod), 179
nrms_sets (fortran variable in module output_mod), 276
nrotic (fortran variable in module init_fields), 191
nrotma (fortran variable in module init_fields), 190
nrp (fortran variable in module truncation), 154
nrpgeos (fortran variable in module truncation), 154
nrstart (fortran variable in module radial_data), 180
nrstartmag (fortran variable in module radial_data), 180
nrstop (fortran variable in module radial_data), 180
nrstopmag (fortran variable in module radial_data), 180
nsmax (fortran variable in module omega), 354
nsmaxa (fortran variable in module output_data), 174
nthetabs (fortran variable in module blocking), 269
nthreads (fortran variable in module parallel_mod), 179
ntomovsets (fortran variable in module output_mod), 275
ntormssets (fortran variable in module output_mod), 275
ntosets (fortran variable in module output_mod), 275
ntpotsets (fortran variable in module output_mod), 274
num_param (module), 155
nvarcond (fortran variable in module physical_parameters), 160
nvardiff (fortran variable in module physical_parameters), 160
nvareps (fortran variable in module physical_parameters), 160
nvarvisc (fortran variable in module physical_parameters), 160
nvpotsets (fortran variable in module output_mod), 275
nzmaxa (fortran variable in module output_data), 172
nzmaxs (fortran variable in module egeos_mod), 348
nzmaxs (fortran variable in module outto_mod), 344

O

o_r_ic (fortran variable in module radial_functions), 197
o_r_ic2 (fortran variable in module radial_functions), 196
o_sin_theta (fortran variable in module horizontal_data), 199
o_sin_theta_e2 (fortran variable in module horizontal_data), 200
o_sr (fortran variable in module physical_parameters), 159
ohmdissr (fortran variable in module power), 290

ohmlossfac (fortran variable in module physical_parameters), 159
omega (module), 353
omega_ic (fortran variable in module fields), 166
omega_ic1 (fortran variable in module init_fields), 190
omega_ic2 (fortran variable in module init_fields), 190
omega_ma (fortran variable in module fields), 167
omega_ma1 (fortran variable in module init_fields), 191
omega_ma2 (fortran variable in module init_fields), 191
omegaosz_ic1 (fortran variable in module init_fields), 190
omegaosz_ic2 (fortran variable in module init_fields), 190
omegaosz_ma1 (fortran variable in module init_fields), 191
omegaosz_ma2 (fortran variable in module init_fields), 191
one (fortran variable in module constants), 177
openfiles() (fortran subroutine in module output_data), 176
opm (fortran variable in module physical_parameters), 158
opr (fortran variable in module physical_parameters), 160
or1 (fortran variable in module radial_functions), 197
or2 (fortran variable in module radial_functions), 197
or3 (fortran variable in module radial_functions), 197
or4 (fortran variable in module radial_functions), 197
orho1 (fortran variable in module radial_functions), 196
orho2 (fortran variable in module radial_functions), 196
osints (fortran variable in module egeos_mod), 347
osints (fortran variable in module outpv3), 350
osints (fortran variable in module outto_mod), 344
osn1 (fortran variable in module horizontal_data), 199
osn2 (fortran variable in module horizontal_data), 199
osq4pi (fortran variable in module constants), 177
otemp1 (fortran variable in module radial_functions), 196
out_coeff (module), 314
out_dtb_frame (module), 334
out_movie (module), 306
out_movie_ic (module), 313
outmisc() (fortran subroutine in module outmisc_mod), 283
outmisc_mod (module), 282
outomega() (fortran subroutine in module omega), 354
outp (fortran variable in module precision_mod), 152
outpar() (fortran subroutine in module outpar_mod), 288
outpar_mod (module), 287
outperppar() (fortran subroutine in module outpar_mod), 289
output() (fortran subroutine in module nonlinear_lm_mod), 231
output() (fortran subroutine in module output_mod), 276
output_data (module), 170
output_mod (module), 272
outpv() (fortran subroutine in module outpv3), 350
outpv3 (module), 349
outrot (module), 284
outto() (fortran subroutine in module outto_mod), 344
outto_mod (module), 343

P

p (fortran variable in module fields), 166
p0mat (fortran variable in module matrices), 169
p0pivot (fortran variable in module matrices), 169
p_ave (fortran variable in module fields_average_mod), 319
p_ave_global (fortran variable in module fields_average_mod), 319
p_lmloc (fortran variable in module fields), 165
p_lmloc_container (fortran variable in module fields), 166
p_rloc (fortran variable in module fields), 165
p_rloc_container (fortran variable in module fields), 166
padvasrms (fortran variable in module dtb_mod), 329
padvlm (fortran variable in module dtb_mod), 329
padvlm_rloc (fortran variable in module dtb_mod), 330
padvlmic (fortran variable in module dtb_mod), 330
padvlmic_lmloc (fortran variable in module dtb_mod), 330
padvrms (fortran variable in module dtb_mod), 330
par_file (fortran variable in module output_data), 172
parallel() (fortran subroutine in module parallel_mod), 180
parallel_mod (module), 179
pdifasrms (fortran variable in module dtb_mod), 329
pdiflm (fortran variable in module dtb_mod), 330
pdiflm_lmloc (fortran variable in module dtb_mod), 330
pdiflmic (fortran variable in module dtb_mod), 330
pdiflmic_lmloc (fortran variable in module dtb_mod), 330
pdifrms (fortran variable in module dtb_mod), 329
peaks (fortran variable in module physical_parameters), 159
perppar_file (fortran variable in module output_data), 173
phi (fortran variable in module horizontal_data), 200
phideravg() (in module magic.libmagic), 147
phis (fortran variable in module physical_parameters), 160
physical_parameters (module), 157
pi (fortran variable in module constants), 177
plf2hint (fortran variable in module rms), 323
plfrmsl_sd (fortran variable in module rms), 321
plfrmsl_ta (fortran variable in module rms), 322
plfrmssd (fortran variable in module rms), 322
plm (fortran variable in module horizontal_data), 199
plm_theta() (fortran subroutine in module plms_theta), 241
plm_thetaas() (fortran subroutine in module plms_theta), 242
plms (fortran variable in module egeos_mod), 348
plms (fortran variable in module outpv3), 350
plms (fortran variable in module outto_mod), 344

plms_theta (module), **241**
plmz (fortran variable in module outpv3), **350**
plot() (magic.bLayers.BLayers method), 137
plot() (magic.Butterfly method), 143
plot() (magic.coeff.MagicCoeffCmb method), 130
plot() (magic.MagicRadial method), 122
plot() (magic.MagicSpectrum method), 122
plot() (magic.MagicSpectrum2D method), 123
plot() (magic.MagicTs method), 120
plot() (magic.Movie method), 129
plot() (magic.ThetaHeat method), 138
plot() (magic.TOMovie method), 132
plotAvg() (magic.CompSims method), 133
plotAvg() (magic.MagicRSpec method), 131
plotEmag() (magic.CompSims method), 133
plotEquat() (magic.CompSims method), 133
plotFlux() (magic.CompSims method), 133
plotSurf() (magic.CompSims method), 133
plotTs() (magic.CompSims method), 133
plotZonal() (magic.CompSims method), 133
polind (fortran variable in module physical_parameters), **158**
power (module), **289**
power_file (fortran variable in module output_data), **171**
pr (fortran variable in module physical_parameters), **159**
pre (fortran variable in module updatewp_mod), **210**
pre2hint (fortran variable in module rms), **323**
precalc() (fortran subroutine in module precalculations), **194**
precalctimes() (fortran subroutine in module precalculations), **194**
precalculations (module), **193**
precision_mod (module), **152**
prermsl_sd (fortran variable in module rms), **323**
prermsl_ta (fortran variable in module rms), **322**
prermssd (fortran variable in module rms), **322**
prmag (fortran variable in module physical_parameters), **160**
progressbar() (in module magic.libmagic), 147
pscale (fortran variable in module num_param), **157**
pstrasrms (fortran variable in module dtb_mod), **330**
pstrlm (fortran variable in module dtb_mod), **330**
pstrlm_rloc (fortran variable in module dtb_mod), **330**
pstrrms (fortran variable in module dtb_mod), **330**

R

r (fortran variable in module radial_functions), **196**
r2lm_redist() (fortran subroutine in module communications), **185**
r2lo_redist() (fortran subroutine in module communications), **185**
r_cmb (fortran variable in module radial_functions), **196**
r_cut_model (fortran variable in module physical_parameters), **159**
r_ic (fortran variable in module radial_functions), **198**
r_icb (fortran variable in module radial_functions), **197**
r_lcr (fortran variable in module physical_parameters), **160**
r_lm_gather_type (fortran variable in module communications), **182**
r_lm_gather_type_lm_end (fortran variable in module communications), **183**
r_request (fortran variable in module communications), **183**
r_surface (fortran variable in module radial_functions), **197**
r_transfer_type (fortran variable in module communications), **183**
r_transfer_type_cont (fortran variable in module communications), **183**
r_transfer_type_nr_end (fortran variable in module communications), **183**
r_transfer_type_nr_end_cont (fortran variable in module communications), **183**
ra (fortran variable in module physical_parameters), **159**
radial() (fortran subroutine in module radial_functions), **198**
radial_data (module), **180**
radial_der (module), **259**
radial_der_even (module), **263**
radial_functions (module), **195**
radial_spectra (module), **345**
radialloop (module), **220**
radialloopg() (fortran subroutine in module radialloop), **221**
radratio (fortran variable in module physical_parameters), **159**
random() (fortran function in module useful), **364**
rank_bn (fortran variable in module parallel_mod), **179**
rank_with_l1m0 (fortran variable in module parallel_mod), **179**
rascaled (fortran variable in module physical_parameters), **160**
rbpspec() (fortran subroutine in module radial_spectra), **346**
rbrspec() (fortran subroutine in module radial_spectra), **346**
rc (fortran variable in module rms), **322**
rcut (fortran variable in module output_data), **175**
rdea (fortran variable in module output_data), **175**
rderavg() (in module magic.libmagic), 147
readcheckpoints (module), **358**
readhdf5_attr_dble() (fortran subroutine in module hdf5helpers), **368**
readhdf5_attr_int() (fortran subroutine in module hdf5helpers), **369**
readhdf5_attribute (fortran variable in module hdf5helpers), **368**

readnamelists() (fortran subroutine in module namelists), 187
readstartfields() (fortran subroutine in module readcheckpoints), 359
rearrangeLat() (magic.MagicGraph method), 124
rgrav (fortran variable in module radial_functions), 197
rho0 (fortran variable in module radial_functions), 197
rho_ratio_ic (fortran variable in module physical_parameters), 158
rho_ratio_ma (fortran variable in module physical_parameters), 159
rhs1 (fortran variable in module updateb_mod), 219
rhs1 (fortran variable in module updates_mod), 216
rhs1 (fortran variable in module updatewp_mod), 210
rhs1 (fortran variable in module updatez_mod), 213
rhs2 (fortran variable in module updateb_mod), 219
rint() (fortran function in module integration), 267
rint_r() (fortran function in module integration), 267
rintic() (fortran function in module integration), 267
riteration_mod (module), 223
riterthetablocking_mod (module), 224
riterthetablocking_openmp_mod (module), 226
rmmean (fortran variable in module output_mod), 276
rmmeanr (fortran variable in module outpar_mod), 288
rms (module), 320
rms_helpers (module), 324
rolmean (fortran variable in module output_mod), 275
rolmeanr (fortran variable in module outpar_mod), 288
rolmeanru2 (fortran variable in module outpar_mod), 288
rot_file (fortran variable in module output_data), 173
rrmp (fortran variable in module bext), 176
rst_file (fortran variable in module output_data), 171
rst_mpi_fh (fortran variable in module output_data), 172
rstrm (fortran variable in module outto_mod), 344
runid (fortran variable in module output_data), 171
runtime (fortran variable in module num_param), 157
runtimelimit (fortran variable in module num_param), 156
runtimestart (fortran variable in module num_param), 157
rz (fortran variable in module egeos_mod), 348
rz (fortran variable in module outpv3), 350
rz (fortran variable in module outto_mod), 344

S

s (fortran variable in module fields), 167
s0mat (fortran variable in module matrices), 169
s0mat_fac (fortran variable in module matrices), 169
s0pivot (fortran variable in module matrices), 169
s_ave (fortran variable in module fields_average_mod), 319
s_ave_global (fortran variable in module fields_average_mod), 319
s_bot (fortran variable in module init_fields), 190
s_cond() (fortran subroutine in module init_fields), 192
s_lmloc (fortran variable in module fields), 165
s_lmloc_container (fortran variable in module fields), 165
s_request (fortran variable in module communications), 183
s_rloc (fortran variable in module fields), 166
s_rloc_container (fortran variable in module fields), 166
s_top (fortran variable in module init_fields), 191
s_transfer_type (fortran variable in module communications), 182
s_transfer_type_cont (fortran variable in module communications), 182
s_transfer_type_nr_end (fortran variable in module communications), 183
s_transfer_type_nr_end_cont (fortran variable in module communications), 182
safeclose() (fortran subroutine in module useful), 366
safeopen() (fortran subroutine in module useful), 365
scale_b (fortran variable in module init_fields), 190
scale_s (fortran variable in module init_fields), 190
scale_v (fortran variable in module init_fields), 190
scanDir() (in module magic.libmagic), 147
scatter_from_rank0_to_lo() (fortran subroutine in module communications), 184
sdens (fortran variable in module output_data), 171
sderavg() (in module magic.libmagic), 148
selectField() (in module magic.libmagic), 148
sendvals_to_rank0() (fortran subroutine in module outrot), 286
set_steering_variables() (fortran subroutine in module riteration_mod), 224
set_thetablocking() (fortran subroutine in module riterthetablocking_mod), 225
set_zero() (fortran subroutine in module dtb_arrays_mod), 333
set_zero() (fortran subroutine in module nonlinear_lm_mod), 231
set_zero() (fortran subroutine in module to_arrays_mod), 343
sgefa() (fortran subroutine in module algebra), 258
sgesl() (fortran subroutine in module algebra), 258
sigma (fortran variable in module radial_functions), 198
sigma_ratio (fortran variable in module physical_parameters), 159
sin36 (fortran variable in module constants), 178
sin60 (fortran variable in module constants), 177
sin72 (fortran variable in module constants), 178
sintheta (fortran variable in module horizontal_data), 200
sizelmb (fortran variable in module blocking), 269
sizelmb2 (fortran variable in module blocking), 269
sizeof_character (fortran variable in module precision_mod), 153
sizeof_def_complex (fortran variable in module precision_mod), 153

sizeof_def_real (fortran variable in module precision_mod), **153**
sizeof_integer (fortran variable in module precision_mod), **153**
sizeof_out_real (fortran variable in module precision_mod), **153**
sizerb (fortran variable in module blocking), **269**
sizethetab (fortran variable in module blocking), **269**
sizethetabi (fortran variable in module blocking), **269**
slice() (magic.cyl.Cyl method), 140
slice() (magic.Surf method), 126
slopestrat (fortran variable in module physical_parameters), **160**
smat (fortran variable in module matrices), **169**
smat_fac (fortran variable in module matrices), **169**
smeanr (fortran variable in module outpar_mod), **288**
sn2 (fortran variable in module horizontal_data), **199**
sn_sub_map (fortran variable in module blocking), **269**
spectra (module), **291**
spectrum() (fortran subroutine in module spectra), **294**
spectrum_average() (fortran subroutine in module spectra), **293**
spectrum_temp() (fortran subroutine in module spectra), **294**
spectrum_temp_average() (fortran subroutine in module spectra), **294**
sph2cart_scal() (in module magic.graph2vtk), 134
sph2cart_vec() (in module magic.graph2vtk), 135
sph2cyl() (in module magic.cyl), 141
sph2cyl_plane() (in module magic.cyl), 141
spivot (fortran variable in module matrices), **169**
sq4pi (fortran variable in module constants), **178**
sric_file (fortran variable in module output_data), **174**
srma_file (fortran variable in module output_data), **175**
st_map (fortran variable in module blocking), **269**
st_sub_map (fortran variable in module blocking), **269**
start_fields (module), **187**
start_file (fortran variable in module init_fields), **190**
step_time() (fortran subroutine in module step_time_mod), **203**
step_time_mod (module), **201**
store() (fortran subroutine in module storecheckpoints), **363**
store_fields_3d() (fortran subroutine in module out_movie), **311**
store_fields_p() (fortran subroutine in module out_movie), **309**
store_fields_r() (fortran subroutine in module out_movie), **309**
store_fields_sur() (fortran subroutine in module out_movie), **308**
store_fields_t() (fortran subroutine in module out_movie), **310**
store_movie_frame() (fortran subroutine in module out_movie), **307**
store_movie_frame_ic() (fortran subroutine in module out_movie_ic), **313**
store_pot_mod (module), **316**
storecheckpoints (module), **362**
storepot() (fortran subroutine in module store_pot_mod), **316**
storepotw() (fortran subroutine in module store_pot_mod), **317**
str2dble() (fortran subroutine in module charmanip), **367**
strat (fortran variable in module physical_parameters), **158**
strm (fortran variable in module outto_mod), **344**
subtime() (fortran subroutine in module timing), **206**
Surf (class in magic), 124
surf() (magic.cyl.Cyl method), 140
surf() (magic.Surf method), 126
surf_cmb (fortran variable in module constants), **177**
svar (fortran variable in module outpar_mod), **288**
symmetrize() (in module magic.libmagic), 148

T

t2_ave (fortran variable in module spectra), **293**
t_ave (fortran variable in module spectra), **293**
t_bpot (fortran variable in module output_data), **175**
t_bpot_start (fortran variable in module output_data), **171**
t_bpot_stop (fortran variable in module output_data), **175**
t_cmb (fortran variable in module output_data), **172**
t_cmb_start (fortran variable in module output_data), **173**
t_cmb_stop (fortran variable in module output_data), **173**
t_graph (fortran variable in module output_data), **172**
t_graph_start (fortran variable in module output_data), **172**
t_graph_stop (fortran variable in module output_data), **174**
t_icb2_ave (fortran variable in module spectra), **292**
t_icb_ave (fortran variable in module spectra), **292**
t_log (fortran variable in module output_data), **175**
t_log_start (fortran variable in module output_data), **173**
t_log_stop (fortran variable in module output_data), **172**
t_movie (fortran variable in module output_data), **174**
t_movie_start (fortran variable in module output_data), **171**
t_movie_stop (fortran variable in module output_data), **175**
t_movies (fortran variable in module movie_data), **299**
t_pot (fortran variable in module output_data), **174**
t_pot_start (fortran variable in module output_data), **174**
t_pot_stop (fortran variable in module output_data), **175**
t_r_field (fortran variable in module output_data), **174**
t_r_field_start (fortran variable in module output_data), **173**

t_r_field_stop (fortran variable in module output_data), 173
t_r_file (fortran variable in module output_data), 171
t_rst (fortran variable in module output_data), 172
t_rst_start (fortran variable in module output_data), 171
t_rst_stop (fortran variable in module output_data), 171
t_spec (fortran variable in module output_data), 171
t_spec_start (fortran variable in module output_data), 175
t_spec_stop (fortran variable in module output_data), 175
t_to (fortran variable in module output_data), 173
t_to_start (fortran variable in module output_data), 172
t_to_stop (fortran variable in module output_data), 175
t_tomovie (fortran variable in module output_data), 175
t_tomovie_start (fortran variable in module output_data), 174
t_tomovie_stop (fortran variable in module output_data), 173
t_toz (fortran variable in module output_data), 174
t_toz_start (fortran variable in module output_data), 171
t_toz_stop (fortran variable in module output_data), 173
t_tpot (fortran variable in module output_data), 174
t_tpot_start (fortran variable in module output_data), 173
t_tpot_stop (fortran variable in module output_data), 174
t_vpot (fortran variable in module output_data), 173
t_vpot_start (fortran variable in module output_data), 172
t_vpot_stop (fortran variable in module output_data), 175
tadvasrms (fortran variable in module dtb_mod), 330
tadvlm (fortran variable in module dtb_mod), 330
tadvlm_rloc (fortran variable in module dtb_mod), 329
tadvlmic (fortran variable in module dtb_mod), 329
tadvlmic_lmloc (fortran variable in module dtb_mod), 329
tadvrlm (fortran variable in module dtb_mod), 330
tadvrlm_rloc (fortran variable in module dtb_mod), 330
tadvrms (fortran variable in module dtb_mod), 330
tag (fortran variable in module output_data), 175
tdifasrms (fortran variable in module dtb_mod), 330
tdiflm (fortran variable in module dtb_mod), 330
tdiflm_lmloc (fortran variable in module dtb_mod), 330
tdiflmic (fortran variable in module dtb_mod), 330
tdiflmic_lmloc (fortran variable in module dtb_mod), 330
tdifrms (fortran variable in module dtb_mod), 329
temp0 (fortran variable in module radial_functions), 197
temp_gather_lo (fortran variable in module communications), 183
temp_r2lo (fortran variable in module communications), 183
tend (fortran variable in module num_param), 156
theta (fortran variable in module horizontal_data), 200
theta_ord (fortran variable in module horizontal_data), 200
thetaderavg() (in module magic.libmagic), 149
ThetaHeat (class in magic), 138
thetas (fortran variable in module physical_parameters), 159
third (fortran variable in module constants), 178
three (fortran variable in module constants), 177
time2ms() (fortran function in module timing), 206
timenormlog (fortran variable in module output_mod), 275
timenormrms (fortran variable in module output_mod), 275
timepassedlog (fortran variable in module output_mod), 275
timepassedrms (fortran variable in module output_mod), 275
timestart (fortran variable in module num_param), 156
timing (module), 205
tipdipole (fortran variable in module init_fields), 190
tmagcon (fortran variable in module physical_parameters), 160
to_arrays_mod (module), 342
to_gather_rloc_on_rank0() (fortran subroutine in module torsional_oscillations), 340
to_helpers (module), 340
tomeasrms (fortran variable in module dtb_mod), 330
tomega_ic1 (fortran variable in module init_fields), 190
tomega_ic2 (fortran variable in module init_fields), 190
tomega_ma1 (fortran variable in module init_fields), 190
tomega_ma2 (fortran variable in module init_fields), 190
tomelm (fortran variable in module dtb_mod), 330
tomelm_rloc (fortran variable in module dtb_mod), 330
tomerlm (fortran variable in module dtb_mod), 330
tomerlm_rloc (fortran variable in module dtb_mod), 330
tomerms (fortran variable in module dtb_mod), 330
TOMovie (class in magic), 131
topcond (fortran variable in module radial_functions), 196
tops (fortran variable in module init_fields), 190
torsional_oscillations (module), 336
transform_to_grid_space() (fortran subroutine in module riterthetablocking_mod), 226
transform_to_lm_space() (fortran subroutine in module riterthetablocking_mod), 226
transportproperties() (fortran subroutine in module radial_functions), 198
truncation (module), 153
tscale (fortran variable in module num_param), 157
tshift_ic1 (fortran variable in module init_fields), 191
tshift_ic2 (fortran variable in module init_fields), 191
tshift_ma1 (fortran variable in module init_fields), 190
tshift_ma2 (fortran variable in module init_fields), 190
tstrasrms (fortran variable in module dtb_mod), 330
tstrlm (fortran variable in module dtb_mod), 329
tstrlm_rloc (fortran variable in module dtb_mod), 330
tstrrlm (fortran variable in module dtb_mod), 330
tstrrlm_rloc (fortran variable in module dtb_mod), 330
tstrrms (fortran variable in module dtb_mod), 330

two (fortran variable in module constants), 178

U

u_square_file (fortran variable in module output_data), 175
uhmeanr (fortran variable in module outpar_mod), 288
ulm (fortran variable in module lmloop_data), 181
ulmmag (fortran variable in module lmloop_data), 181
unknown_type (fortran type in module communications), 182
unknown_type (fortran type in module cosine_transform_even), 239
unknown_type (fortran type in module cosine_transform_odd), 237
unknown_type (fortran type in module dtb_arrays_mod), 333
unknown_type (fortran type in module leg_helper_mod), 243
unknown_type (fortran type in module nonlinear_lm_mod), 230
unknown_type (fortran type in module riteration_mod), 223
unknown_type (fortran type in module riterthetablocking_mod), 225
unknown_type (fortran type in module riterthetablocking_openmp_mod), 227
unknown_type (fortran type in module to_arrays_mod), 342
updateb() (fortran subroutine in module updateb_mod), 219
updateb_mod (module), 217
updates() (fortran subroutine in module updates_mod), 216
updates_ala() (fortran subroutine in module updates_mod), 216
updates_mod (module), 215
updatewp() (fortran subroutine in module updatewp_mod), 211
updatewp_mod (module), 209
updatez() (fortran subroutine in module updatez_mod), 213
updatez_mod (module), 212
useful (module), 364

V

v2as (fortran variable in module torsional_oscillations), 337
v2as_rloc (fortran variable in module torsional_oscillations), 337
v_r_file (fortran variable in module output_data), 172
v_rigid_boundary() (fortran subroutine in module nonlinear_bcs), 234
visc (fortran variable in module radial_functions), 197
vischeatfac (fortran variable in module physical_parameters), 158
vol_ic (fortran variable in module constants), 178
vol_oc (fortran variable in module constants), 178
vorold (fortran variable in module outpv3), 350
vpm (fortran variable in module outto_mod), 344
vscale (fortran variable in module num_param), 157

W

w (fortran variable in module fields), 167
w_ave (fortran variable in module fields_average_mod), 319
w_ave_global (fortran variable in module fields_average_mod), 319
w_lmloc (fortran variable in module fields), 166
w_lmloc_container (fortran variable in module fields), 166
w_rloc (fortran variable in module fields), 166
w_rloc_container (fortran variable in module fields), 166
walltime() (fortran subroutine in module timing), 206
widths (fortran variable in module physical_parameters), 160
worka (fortran variable in module updateb_mod), 218
worka (fortran variable in module updates_mod), 216
worka (fortran variable in module updatewp_mod), 210
worka (fortran variable in module updatez_mod), 213
workb (fortran variable in module updateb_mod), 219
workb (fortran variable in module updates_mod), 216
workb (fortran variable in module updatewp_mod), 210
workb (fortran variable in module updatez_mod), 213
workc (fortran variable in module updatez_mod), 213
wpass2jw() (fortran subroutine in module fft), 255
wpass3jw() (fortran subroutine in module fft), 256
wpass4jw() (fortran subroutine in module fft), 256
wpass5jw() (fortran subroutine in module fft), 256
wplm (fortran variable in module horizontal_data), 200
wpmat (fortran variable in module matrices), 169
wpmat_fac (fortran variable in module matrices), 169
wppivot (fortran variable in module matrices), 169
write_bcmb() (fortran subroutine in module out_coeff), 314
write_coeff_r() (fortran subroutine in module out_coeff), 315
write_dataset() (fortran subroutine in module hdf5helpers), 368
write_dtb_frame() (fortran subroutine in module out_dtb_frame), 334
write_movie_frame() (fortran subroutine in module out_movie), 308
write_rot() (fortran subroutine in module outrot), 285
writehdf5_attr_dble() (fortran subroutine in module hdf5helpers), 369
writehdf5_attr_int() (fortran subroutine in module hdf5helpers), 369

writehdf5_attribute (fortran variable in module hdf5helpers), **368**
writeinfo() (fortran subroutine in module precalculations), **194**
writenamelists() (fortran subroutine in module namelists), **187**
writetime() (fortran subroutine in module timing), **207**
writeVpEq() (in module magic.libmagic), 149
writeVTI() (magic.graph2vtk.Graph2Vtk method), 134
writeVTS() (magic.graph2vtk.Graph2Vtk method), 134

Y

y10_norm (fortran variable in module constants), **177**
y11_norm (fortran variable in module constants), **178**

Z

z (fortran variable in module fields), **166**
z10mat (fortran variable in module matrices), **169**
z10mat_fac (fortran variable in module matrices), **169**
z10pivot (fortran variable in module matrices), **169**
z_ave (fortran variable in module fields_average_mod), **318**
z_ave_global (fortran variable in module fields_average_mod), **319**
z_lmloc (fortran variable in module fields), **166**
z_lmloc_container (fortran variable in module fields), **166**
z_rloc (fortran variable in module fields), **165**
z_rloc_container (fortran variable in module fields), **167**
zavg() (in module magic.cyl), 141
zdens (fortran variable in module output_data), **172**
zderavg() (in module magic.libmagic), 149
zero (fortran variable in module constants), **177**
zero_tolerance (fortran variable in module algebra), **257**
zerorms() (fortran subroutine in module rms), **323**
zmat (fortran variable in module matrices), **169**
zmat_fac (fortran variable in module matrices), **169**
zpivot (fortran variable in module matrices), **169**
zz (fortran variable in module egeos_mod), **348**
zz (fortran variable in module outto_mod), **344**