MAGIC Documentation

Release 5.2

The MAGIC dev team

October 16, 2015

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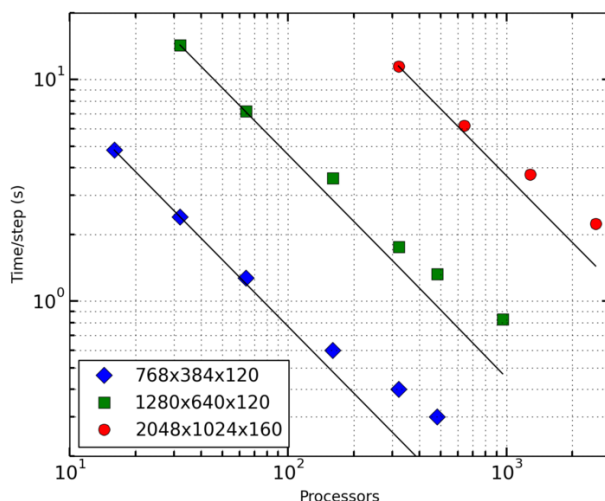
Fig. 1.1: Mean walltime of the MagIC code on the supercomputer stampede versus number of CPUs for a dynamo model computed at three different numerical resolutions (N_ϕ, N_θ, N_r). The solid black lines show the ideal scalings.

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)

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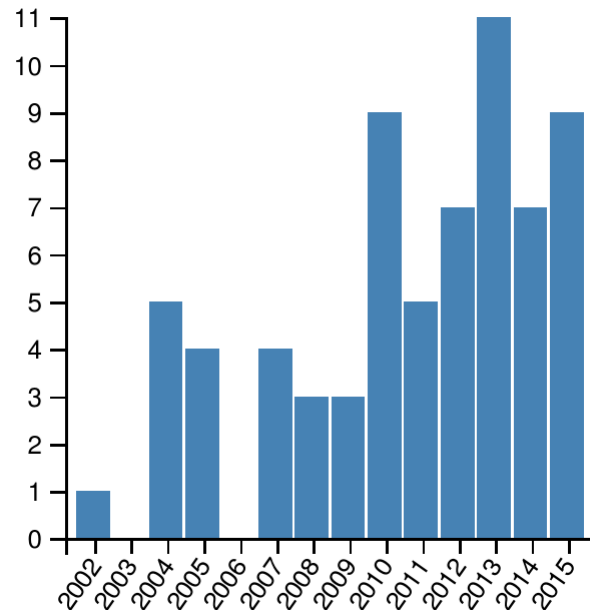
Fig. 1.2: Number of peer-reviewed publications produced using **MagIC**

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

Go to the directory where the source files of MagIC are contained

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

and edit the `Makefile` there to specify your specific compiler options.

2.3.1 Makefile options

Select compiler

Set a suitable compiler in the first line of the `Makefile`: `COMPILER = <compiler_phrase>`. The options are `intel`, `gnu` or `amd` - depending on your available compilers.

List of default compilers

Compiler Option	Normal	With MPI
intel	ifort, icc	mpiifort, mpiicc
gnu	gfortran, gcc	mpif90, mpicc
amd	openf95	

Select compiling 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

- `PRODRUN` Set it to `yes` for production run, `no` for debugging.

- `USE_MPI` Set to `yes` to use MPI, set it to `no` if you want a serial version of the code .

- `OPENMP` Set it to `yes` to use the hybrid version of the code, or to `no` for a pure MPI (or serial) version.

- `DEBUG` Set to `all` to enable the full debug flags. *While running in debugging mode, set* `PRODRUN` *to* `no`.

- `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_MKL` Set to `yes` if you have Intel MKL installed and want to use it for matrix operations.

- `USE_HDF5` Set to `yes` if you want the restart file to be written in the HDF5 format

Architecture (Intel compilers only)

If you're using intel compilers and if your computer is capable of following specific intel instruction sets (sse3 or AVX), then the `Makefile` automatically should automatically detects and sets `FFLAG_ARCH_OPT = -xsse3` or `FFLAG_ARCH_OPT = -xAVX` under intel compiler options.

MPI_INCPATH

This sets the path for your mpi header file `mpif.h` . The path depends on the computer. For PCs, this is commonly `/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 since the `mpi.mod` file is usually find the standard `$PATH`.

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 compilers with mpi implementation.

2.3.2 Compiling the code

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 `OPENMP=no`):

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

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

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

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

```
$ export OMP_NUM_THREAD = <n_omp>
$ mpiexec -n <n_mpi> ./magic.exe input.nml
```

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

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. The conservation of mass is expressed by the continuity equation:

$$\frac{\partial \rho}{\partial t} + \boldsymbol{\nabla} \cdot \rho \boldsymbol{u} = 0, \tag{3.1}$$

The conservation of momentum by

$$\rho \left(\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \boldsymbol{\nabla} \, \boldsymbol{u} \right) = -\boldsymbol{\nabla} p + \frac{1}{\mu_0} (\boldsymbol{\nabla} \times \boldsymbol{B}) \times \boldsymbol{B} + \rho \boldsymbol{g} - 2\rho \boldsymbol{\Omega} \times \boldsymbol{u} + \boldsymbol{\nabla} \cdot \mathsf{S}, \tag{3.2}$$

where S corresponds to the rate-of-strain tensor given by:

$$S_{ij} = 2\nu\rho \left[e_{ij} - \frac{1}{3} \delta_{ij} \, \boldsymbol{\nabla} \cdot \boldsymbol{u} \right],$$

$$e_{ij} = \frac{1}{2} \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right).$$

Concerning the energy equation, several forms are possible (using internal energy, temperature or entropy). Here we use entropy s as the main variable, which leads to:

$$\rho T \left(\frac{\partial s}{\partial t} + \boldsymbol{u} \cdot \boldsymbol{\nabla} s \right) = \boldsymbol{\nabla} \cdot (K \boldsymbol{\nabla} T) + \Phi_\nu + \lambda \left(\boldsymbol{\nabla} \times \boldsymbol{B} \right)^2, \tag{3.3}$$

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

$$\frac{\partial \boldsymbol{B}}{\partial t} = \boldsymbol{\nabla} \times (\boldsymbol{u} \times \boldsymbol{B} - \lambda \, \boldsymbol{\nabla} \times \boldsymbol{B}). \tag{3.4}$$

In those equations, the symbols $\boldsymbol{u}$, $\boldsymbol{B}$, p and s correspond to velocity, magnetic field, pressure and entropy. $\boldsymbol{g}$ corresponds to gravity and ρ to density. λ is the magnetic diffusivity, μ_0 the magnetic permeability, ν the kinematic viscosity and K the thermal conductivity.

3.1 The anelastic approximation

MagIC relies on the anelastic approximation of the Navier Stokes equations. There are different flavours of this approximation but they all assume that:
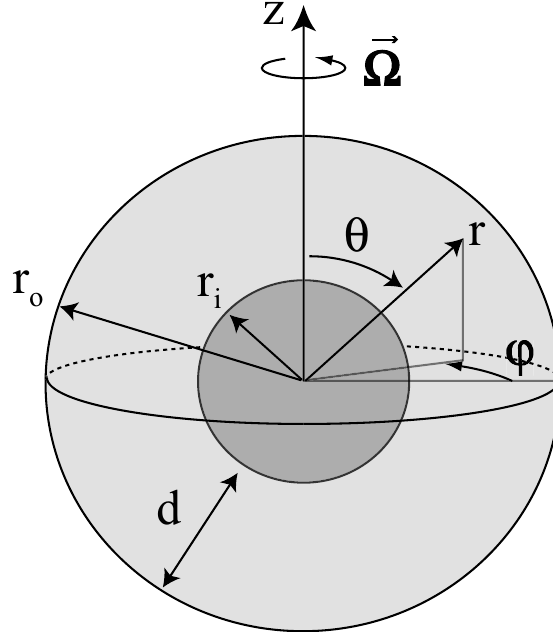
Fig. 3.1: Sketch of the spherical shell model and its system of coordinate.

1. the departures from the thermodynamic state variables due to convection are small with respect to the reference state,

2. the short-period acoustic waves are filtered-out.

This second assumption implies that larger numerical timesteps will be allowed, since the typical timescale of the fast acoustic waves is typically much slower than the convective turnover time. The strict elimination of the acoustic waves formally means

$$\frac{\partial \rho'}{\partial t} = 0$$

in the continuity equation, where ρ' corresponds here to the density perturbations of the reference state.

The reference state is a background state against which perturbations are described. In MagIC, this background state is assumed to only depends on one spatial variable (radius), but in general it may as well be explicitly time-dependent. Each thermodynamic variable f is then expressed as a sum of a spherically-symmetric time-independent quantity $\tilde{f}$ and a fluctuating quantity f':

$$f(r, \theta, \phi, t) = \tilde{f}(r) + f'(r, \theta, \phi, t)$$

This separation of variables is then introduced in the set of equations (3.1)-(3.4) to perform a formal **scale analysis**, taking into account that $f'/\tilde{f} = \epsilon \ll 1$. The variables are then expanded in power series of ϵ and only the highest order terms are retained.

3.2 An adiabatic reference state

In a vigorously convecting astrophysical of geophysical system (like the convecting interior of a planet or a star), the super-adiabaticity of the fluid is extremely small, since the transport of heat by convective motions is highly efficient. Therefore, **the reference background state can be assumed to be perfectly adiabatic** and obey to the following

equations

$$\frac{d\tilde{T}}{dr} = -\frac{\alpha g \tilde{T}}{c_p},$$ (3.5)

where c_p is the heat capacity and α is expressed by

$$c_p = T\left(\frac{\partial s}{\partial T}\right)_p, \quad \text{and} \quad \alpha = -\frac{1}{\rho}\left(\frac{\partial \rho}{\partial T}\right)_p.$$

At this stage, it becomes convenient to start introducing **non-dimensionalised quantities**. The background quantities (density temperature and transport properties) are non-dimensionalised using their values at the spherical shell outer boundary r_o. The shell thickness $d = r_o - r_i$ is used as the reference lenghtscale. The dimensionless form of Eq. (3.5) then reads:

$$\frac{d\tilde{T}}{dr} = -Di\,\alpha(r)g(r)\tilde{T}(r),$$ (3.6)

where $\tilde{T}$, α and g have been non-dimensionalised using their values at the outer boundary. Di is the dissipation number expressed by

$$Di = \frac{\alpha_o g_o d}{c_p}.$$ (3.7)

Di is a measure of the thermal effects due to compressibility, namely viscous and ohmic heating. Di is also the ratio between two length scales: the thickness of the spherical shell d divided by the temperature scale heights $H_T = -(d\ln T/dr)^{-1}$.

When $Di \ll 1$, the region where convection develops has a nearly constant reference temperature. Since there is no basic temperature stratification, viscous heating (which is the consequence of the thermal stratification due to compressibility) becomes negligible.

Note: The Boussinesq limit can thus be recovered by using $Di \to 0$.

Provided an equation of state is given, it is then possible to integrate Eq. (3.6),to obtain the adiabatic background state.

3.2.1 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:

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

where $\gamma = c_p/c_v$. If we now 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

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

where $N_\rho = \ln(\rho_i/\rho_o)$ 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 MHD equations

One of the assumptions of the anelastic approximation is that the fluctuations due to convection are much smaller than the reference state:

$$\epsilon \simeq \frac{\rho'}{\tilde{\rho}} \simeq \frac{T'}{\tilde{T}} \simeq \frac{p'}{\tilde{p}} \simeq s' \ll 1.$$

In the following, we will treat the equations (3.1)-(3.4) in nondimensional form. There is no unique way to scale the equations and as a consequence different sets of non-dimensional numbers are employed. For convection-driven dynamos, there is four independent control parameters.

We use here the viscous diffusion time d^2/ν_o (where ν_o is the kinematic viscosity at the outer boundary) as a time unit and ν_o/d as the reference velocity. Magnetic field is expressed in units of $\sqrt{\rho_o \mu_0 \lambda_i \Omega}$, where ρ_o is the density at the outer boundary and λ_i is the magnetic diffusivity at the **inner** boundary.

Note: All the transport properties except the magnetic diffusivity are normalised to their values at the outer boundary. The motivation to rather base the reference magnetic diffusivity to the **inner** boundary 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.

This leads to the following sets of dimensionless equations:

$$E \left(\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \boldsymbol{\nabla} \boldsymbol{u} \right) + 2 \boldsymbol{e_z} \times \boldsymbol{u} = -\boldsymbol{\nabla} \left(\frac{p'}{\tilde{\rho}} \right) + \frac{Ra\, E}{Pr} g(r)\, s'\, \boldsymbol{e_r} + \frac{1}{Pm\, \tilde{\rho}} \left(\boldsymbol{\nabla} \times \boldsymbol{B} \right) \times \boldsymbol{B} + \frac{E}{\tilde{\rho}} \boldsymbol{\nabla} \cdot \mathsf{S}, \qquad (3.8)$$

$$\boldsymbol{\nabla} \cdot \tilde{\rho} \boldsymbol{u} = 0, \qquad (3.9)$$

$$\boldsymbol{\nabla} \cdot \boldsymbol{B} = 0, \qquad (3.10)$$

$$\frac{\partial \boldsymbol{B}}{\partial t} = \boldsymbol{\nabla} \times (\boldsymbol{u} \times \boldsymbol{B}) - \frac{1}{Pm} \boldsymbol{\nabla} \times (\lambda(r)\, \boldsymbol{\nabla} \times \boldsymbol{B}). \qquad (3.11)$$

3.3.1 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. (3.3) 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

$$\epsilon \frac{T'/\delta^2}{T/d^2}, \tag{3.12}$$

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 (3.12) 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-dimensionalise Eq. (3.3) 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.

$$\tilde{\rho} \tilde{T} \left(\frac{\partial s'}{\partial t} + \boldsymbol{u} \cdot \boldsymbol{\nabla} s' \right) = \frac{1}{Pr} \boldsymbol{\nabla} \cdot \left(\kappa(r) \tilde{\rho} \tilde{T} \boldsymbol{\nabla} s' \right) + \frac{Pr\,Di}{Ra} \Phi_\nu + \frac{Pr\,Di}{Pm^2\,E\,Ra} \lambda(r) \left(\boldsymbol{\nabla} \times \boldsymbol{B} \right)^2, \tag{3.13}$$

3.3.2 The Boussinesq limits of the equation $Di \to 0$

When the dissipation number $Di \to 0$ then $\tilde{T} = $ cst.. If in addition to that if $\gamma \neq 1$, the density background $\tilde{\rho}$ is also constant.

Note: The peculiar configuration of $\gamma = 1$ corresponds to the so-called zero-Grüneisen limit of the Navier-Stokes equation (or isothermal) and is a special case in which $Di = 0$ but a density background (controlled by N_ρ) is still allowed.

A brief look at Eq. (3.13) then shows than viscous and Ohmic heating will disappear from the entropy equation. Furthermore, temperature and entropy fluctuations become equivalent quantities. If in addition to that we also neglect the possible radial-dependence of the transport properties (electrical conductivity, viscosity and thermal diffusivity), the set of equations (3.8)-(3.13) thus simplifies to the classical Boussinesq set of equations:

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

$$\boldsymbol{\nabla} \cdot \boldsymbol{u} = 0,$$

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

$$\frac{\partial \boldsymbol{B}}{\partial t} = \boldsymbol{\nabla} \times (\boldsymbol{u} \times \boldsymbol{B}) + \frac{1}{Pm} \Delta \boldsymbol{B}.$$

$$\frac{\partial T'}{\partial t} + \boldsymbol{u} \cdot \boldsymbol{\nabla} T' = \frac{1}{Pr} \Delta T'.$$

3.3.3 Dimensionless control parameters

The equations (3.8)-(3.13) are governed by four dimensionless numbers: the Ekman number

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

(3.14)

the Rayleigh number

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

(3.15)

the Prandtl number

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

(3.16)

and the magnetic Prandtl number

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

(3.17)

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

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

(3.18)

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

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 (3.8), is often incorporated into the definition of the Ekman number.

See also:

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

3.3.4 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.

3.4 Boundary conditions and treatment of inner core

3.4.1 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.

3.4.2 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 (3.11) 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.

3.4.3 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.

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

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

The two scalar potentials of a divergence free vector field can be extracted from its radial component and the radial component of its curl:

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

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

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

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 $g_{\ell m}^\star(r) = g_{\ell,-m}(r)$, 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

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

(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

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

(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:

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

(4.12)

Here, we have assumed that the Legendre functions are completely normalised 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

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

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:

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

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:

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

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

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

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

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

with

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

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

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

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 $W_{\ell mn}$:

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

$$(4.20)$$

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.

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

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 $Z_{\ell m n}$:

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

(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 $P_{\ell m n}$:

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

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

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

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 field coefficient reads

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

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

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

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.

W consequently adopt in **MagIC** a mixed implicit/explicit algorithm. Nonlinear and Coriolis terms, collected on the right hand side of equations (4.20), (4.21), (4.22), (4.24), (4.25) and (4.26) are treated explicitly with a second order Adams-Bashforth . Terms collected on the left hand side are time-stepped with an implicit modified Crank-Nicolson algorithm. While the equations are coupled radially, they decouple for all spherical harmonic modes.

Note: The poloidal flow potential (4.20) and the pressure (4.22) are nevertheless coupled for a given spherical harmonic mode.

As an example, we derive the time stepping equation for the poloidal magnetic potential of degree ℓ and order m, denoting the explicit nonlinear term at radial grid point r_k with

$$
D_{k\ell m}(t) = \int d\Omega\, Y_\ell^{m\star}\, \boldsymbol{e_r}\cdot\boldsymbol{D}(t,r_k,\theta,\phi)\;.
$$

After discretization of the partial time derivative, $\partial g_{\ell mn}/\partial t = [g_{\ell mn}(t+\delta t) - g_{\ell mn}(t)]/\delta t$ where δt is the time step, we can formulate the left hand side of (4.25) as a matrix multiplication. The matrices A and G are defined by

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

and

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

where $\mathcal{C}_{nk} = \mathcal{C}_n(r_k)$. The matrices depend on ℓ but not on m. Advancing time from t to $t + \delta t$ is then a matter of solving

$$(A_{kn} + \alpha G_{kn}) \, g_{\ell mn}(t+\delta t) = (A_{kn} - (1-\alpha)G_{kn}) \, g_{\ell mn}(t) + \frac{3}{2} D_{k\ell m}(t) - \frac{1}{2} D_{k\ell m}(t-\delta t) \,.$$

The classical Crank-Nicholson scheme is recovered for $\alpha = 0.5$, but it seems that a slightly larger weight of $\alpha = 0.6$ helps to stabilize the time integration. Since the stability requirements limiting δt will usually change during a computational run, the time step should be adjusted accordingly. The matrix G remains unchanged, but A has to be updated whenever δt is changed. This, in turn, requires a new triangulation of matrix $A_{kn} + \alpha G_{kn}$, which is then stored for subsequent time steps until the next adjustment of δt is in order.

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:

$$\tilde{\rho}\,(\boldsymbol{u}\cdot\boldsymbol{\nabla}\boldsymbol{u}) = \left\{\begin{matrix}\mathcal{A}_r\\\mathcal{A}_\theta\\\mathcal{A}_\phi\end{matrix}\right\} = \left\{\begin{matrix}-\tilde{\rho}\,E\left(u_r\dfrac{\partial u_r}{\partial r} + \dfrac{u_\theta}{r}\dfrac{\partial u_r}{\partial\theta} + \dfrac{u_\phi}{r\sin\theta}\dfrac{\partial u_r}{\partial\phi} - \dfrac{u_\theta^2 + u_\phi^2}{r}\right) + \dfrac{1}{Pm}\left(j_\theta\,B_\phi - j_\phi\,B_\theta\right),\\[1.5em] -\tilde{\rho}\,E\left(u_r\dfrac{\partial u_\theta}{\partial r} + \dfrac{u_\theta}{r}\dfrac{\partial u_\theta}{\partial\theta} + \dfrac{u_\phi}{r\sin\theta}\dfrac{\partial u_\theta}{\partial\phi} + \dfrac{u_r u_\theta}{r} - \dfrac{\cos\theta}{r\sin\theta}u_\phi^2\right) + \dfrac{1}{Pm}\left(j_\phi\,B_r - j_r\,B_\phi\right),\\[1.5em] -\tilde{\rho}\,E\left(u_r\dfrac{\partial u_\phi}{\partial r} + \dfrac{u_\theta}{r}\dfrac{\partial u_\phi}{\partial\theta} + \dfrac{u_\phi}{r\sin\theta}\dfrac{\partial u_\phi}{\partial\phi} + \dfrac{u_r u_\phi}{r} + \dfrac{\cos\theta}{r\sin\theta}u_\theta u_\phi\right) + \dfrac{1}{Pm}\left(j_r\,B_\theta - j_\theta\,B_r\right),\end{matrix}\right\}$$

$$\tag{4.28}$$

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

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

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:

$$\mathcal{N}_{\ell m}^{W} = \frac{2}{r} \left[im \frac{\partial W_\ell^m}{\partial r} - (\ell - 1) c_\ell^m Z_{\ell-1}^m + (\ell + 2) c_{\ell+1}^m Z_{\ell+1}^m \right] + \mathcal{A}_{r\ell}{}^m . \tag{4.29}$$

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 (4.29) 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:

$$\boldsymbol{e_r} \cdot \boldsymbol{\nabla} \times [(2\tilde{\rho}\boldsymbol{u}) \times \boldsymbol{e_z}] = 2\boldsymbol{e_r} \cdot [(\boldsymbol{e_z} \cdot \boldsymbol{\nabla})(\tilde{\rho}\boldsymbol{u})] ,$$

$$= 2 \left[\cos\theta \frac{\partial(\tilde{\rho}u_r)}{\partial r} - \frac{\sin\theta}{r} \frac{\partial(\tilde{\rho}u_r)}{\partial\theta} + \frac{\tilde{\rho}u_\theta \sin\theta}{r} \right] ,$$

$$= 2 \left[-\cos\theta \frac{\partial}{\partial r}(\Delta_H W) + \frac{\sin\theta}{r} \frac{\partial}{\partial\theta}(\Delta_H W) + \frac{\sin\theta}{r^2} \frac{\partial^2 W}{\partial r \partial\theta} + \frac{1}{r^2} \frac{\partial Z}{\partial\phi} \right] .$$

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

$$\boldsymbol{e_r} \cdot \boldsymbol{\nabla} \times [(2\tilde{\rho}\boldsymbol{u}) \times \boldsymbol{e_z}] = \frac{2}{r^2} \left(\vartheta_3 \frac{\partial W}{\partial r} - \frac{1}{r} \vartheta_4 W + \frac{\partial Z}{\partial\phi} \right) . \tag{4.30}$$

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:

$$\vartheta_1 \mathcal{A}p_\ell^m - \frac{\partial\mathcal{A}t_\ell^m}{\partial\phi} . \tag{4.31}$$

Using (4.30) and (4.31), one thus finally gets

$$\begin{aligned} \mathcal{N}_{\ell m}^Z = \frac{2}{r^2} &\left[(\ell - 1)(\ell + 1) c_\ell^m \frac{\partial W_{\ell-1}^m}{\partial r} + \ell(\ell + 2) c_{\ell+1}^m \frac{\partial W_{\ell+1}^m}{\partial r} \right. \\ &\left. - \frac{\ell(\ell - 1)(\ell + 1)}{r} c_\ell^m W_{\ell-1}^m + \frac{\ell(\ell + 1)(\ell + 2)}{r} c_{\ell+1}^m W_{\ell+1}^m + im Z_\ell^m \right] \\ &+ (\ell + 1) c_\ell^m \mathcal{A}p_{\ell-1}^m - \ell c_{\ell+1}^m \mathcal{A}p_{\ell+1}^m - im \mathcal{A}t_\ell^m . \end{aligned} \tag{4.32}$$

See also:

The final calculations of (4.32) 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:

$$\boldsymbol{\nabla}_H \cdot [(2\tilde{\rho}\boldsymbol{u}) \times \boldsymbol{e_z}] = 2\boldsymbol{e_z} \cdot [\boldsymbol{\nabla} \times (\tilde{\rho}\boldsymbol{u})] - \left(\frac{\partial}{\partial r} + \frac{2}{r}\right)[\boldsymbol{e_r} \cdot (2\tilde{\rho}\boldsymbol{u} \times \boldsymbol{e_z})],$$

$$= -2\cos\theta\,\Delta_H Z - 2\sin\theta\left[-\frac{1}{r\sin\theta}\frac{\partial}{\partial\phi}\left(\frac{\partial^2}{\partial r^2} + \Delta_H\right)W + \frac{1}{r}\frac{\partial^2 Z}{\partial r\partial\theta}\right]$$

$$- \left(\frac{\partial}{\partial r} + \frac{2}{r}\right)[2\sin\theta\,\tilde{\rho}u_\phi],$$

$$= 2\left[\frac{1}{r}\left(\Delta_H + \frac{\partial^2}{\partial r^2}\right)\frac{\partial W}{\partial\phi} - \cos\theta\Delta_H Z - \frac{\sin\theta}{r}\frac{\partial^2 Z}{\partial r\partial\theta}\right]$$

$$- \left(\frac{\partial}{\partial r} + \frac{2}{r}\right)\left[\frac{2}{r}\left(\frac{\partial^2 W}{\partial r\partial\phi} - \sin\theta\frac{\partial Z}{\partial\theta}\right)\right],$$

$$= 2\left(\frac{\Delta_H}{r}\frac{\partial W}{\partial\phi} - \frac{1}{r^2}\frac{\partial^2 W}{\partial\phi\partial r} - \cos\theta\Delta_H Z + \frac{\sin\theta}{r^2}\frac{\partial Z}{\partial\theta}\right).$$

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

$$\boldsymbol{\nabla}_H \cdot [(2\tilde{\rho}\boldsymbol{u}) \times \boldsymbol{e_z}] = \frac{2}{r^2}\left(-\frac{L_H}{r}\frac{\partial W}{\partial\phi} - \frac{\partial^2 W}{\partial\phi\partial r} + \vartheta_3\,Z\right). \tag{4.33}$$

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.

$$\vartheta_1\,\mathcal{A}t_\ell^m + \frac{\partial\mathcal{A}p_\ell^m}{\partial\phi}. \tag{4.34}$$

Using (4.33) and (4.34), one thus finally gets

$$\boxed{\begin{aligned}\mathcal{N}_{\ell m}^P = &\frac{2}{r^2}\left[-im\frac{\ell(\ell+1)}{r}W_\ell^m - im\frac{\partial W_\ell^m}{\partial r} + (\ell-1)(\ell+1)\,c_\ell^m\,Z_{\ell-1}^m + \ell(\ell+2)\,c_{\ell+1}^m\,Z_{\ell+1}^m\right]\\ &+ (\ell+1)\,c_\ell^m\,\mathcal{A}t_{\ell-1}^m - \ell\,c_{\ell+1}^m\,\mathcal{A}t_{\ell+1}^m + im\,\mathcal{A}p_\ell^m.\end{aligned}} \tag{4.35}$$

See also:

The final calculations of (4.35) 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 twofolds: (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:

$$
\begin{aligned}
\mathcal{H} =&\frac{Pr\,Di}{Ra}\frac{1}{\tilde{\rho}\tilde{T}}\left[\tilde{\rho}\nu\left\{2\left(\frac{\partial u_r}{\partial r}\right)^2 + 2\left(\frac{1}{r}\frac{\partial u_\theta}{\partial\theta}+\frac{u_r}{r}\right)^2 + 2\left(\frac{1}{r\sin\theta}\frac{\partial u_\phi}{\partial\phi}+\frac{u_r}{r}+\frac{\cos\theta}{r\sin\theta}u_\theta\right)^2\right.\right.\\
&+\left(r\frac{\partial}{\partial r}\left(\frac{u_\theta}{r}\right)+\frac{1}{r}\frac{\partial u_r}{\partial\theta}\right)^2 + \left(r\frac{\partial}{\partial r}\left(\frac{u_\phi}{r}\right)+\frac{1}{r\sin\theta}\frac{\partial u_r}{\partial\phi}\right)^2\\
&+\left.\left(\frac{\sin\theta}{r}\frac{\partial}{\partial\theta}\left(\frac{u_\phi}{\sin\theta}\right)+\frac{1}{r\sin\theta}\frac{\partial u_\theta}{\partial\phi}\right)^2 - \frac{2}{3}\left(\frac{d\ln\tilde{\rho}}{dr}u_r\right)^2\right\}\\
&+\frac{\lambda}{Pm^2\,E}\left\{j_r^2+j_\theta^2+j_\phi^2\right\}\Bigg].
\end{aligned}
\tag{4.36}
$$

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 $\mathcal{US}r_\ell^m$, $\mathcal{US}t_\ell^m$ and $\mathcal{US}p_\ell^m$ are defined.

3. Calculate the colatitude and theta derivatives using the recurrence relations:

$$-\frac{1}{\tilde{\rho}}\left[\frac{1}{r^2}\frac{\partial\mathcal{US}r_\ell^m}{\partial r} + \vartheta_1\mathcal{US}t_\ell^m + \frac{\partial\mathcal{US}p_\ell^m}{\partial\phi}\right].\tag{4.37}$$

Using (4.36) and (4.37), one thus finally gets

$$\boxed{\mathcal{N}_{\ell m}^S = -\frac{1}{\tilde{\rho}}\left[\frac{1}{r^2}\frac{\partial\mathcal{US}r_\ell^m}{\partial r} + (\ell+1)\,c_\ell^m\,\mathcal{US}t_{\ell-1}^m - \ell\,c_{\ell+1}^m\,\mathcal{US}t_{\ell+1}^m + im\mathcal{US}p_\ell^m\right] + \mathcal{H}_\ell^m.}\tag{4.38}$$

See also:

The θ and ϕ derivatives that enter (4.38) 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 $\mathcal{E}_r$, $\mathcal{E}_\theta$ and $\mathcal{E}_\phi$, the three components of the electromotive force $\boldsymbol{u} \times \boldsymbol{B}$:

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

The radial component of the induction term then reads:

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

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

1. Compute the quantities $r^2 \, \mathcal{E}_r$, $\mathcal{E}_\phi/r \sin \theta$ and $\mathcal{E}_\theta/r \sin \theta$ 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 $r^2 \, \mathcal{E}_r$, $\mathcal{E}_\phi/r \sin \theta$ and $\mathcal{E}_\theta/r \sin \theta$ 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 $\mathcal{E}r_\ell^m$, $\mathcal{E}t_\ell^m$ and $\mathcal{E}p_\ell^m$ are defined.

3. Calculate the colatitude and theta derivatives using the recurrence relations:

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

We thus finally get

$$\mathcal{N}_{\ell m}^g = (\ell + 1)\, c_\ell^m \, \mathcal{E}p_{\ell-1}^m - \ell \, c_{\ell+1}^m \, \mathcal{E}p_{\ell+1}^m - im \, \mathcal{E}t_\ell^m \,. \tag{4.39}$$

See also:

The final calculations of (4.39) 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):

$$\mathcal{N}^h = \boldsymbol{e_r} \cdot [\boldsymbol{\nabla} \times \boldsymbol{\nabla} \times (\boldsymbol{u} \times \boldsymbol{B})] = \boldsymbol{e_r} \cdot [\boldsymbol{\nabla} (\boldsymbol{\nabla} \cdot \boldsymbol{\mathcal{E}}) - \Delta \boldsymbol{\mathcal{E}}] \,,$$

$$= \frac{\partial}{\partial r} \left[\frac{1}{r^2} \frac{\partial (r^2 \mathcal{E}_r)}{\partial r} + \frac{1}{r \sin \theta} \frac{\partial (\sin \theta \, \mathcal{E}_\theta)}{\partial \theta} + \frac{1}{r \sin \theta} \frac{\partial \mathcal{E}_\phi}{\partial \phi} \right]$$
$$- \Delta \mathcal{E}_r + \frac{2}{r^2} \left[\mathcal{E}_r + \frac{1}{\sin \theta} \frac{\partial (\sin \theta \, \mathcal{E}_\theta)}{\partial \theta} + \frac{1}{\sin \theta} \frac{\partial \mathcal{E}_\phi}{\partial \phi} \right] \,,$$
$$= \frac{1}{r^2} \frac{\partial}{\partial r} \left[\frac{r}{\sin \theta} \left(\frac{\partial (\sin \theta \, \mathcal{E}_\theta)}{\partial \theta} + \frac{\partial \mathcal{E}_\phi}{\partial \phi} \right) \right] - \Delta_H \, \mathcal{E}_r \,.$$

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:

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

We thus finally get

$$\mathcal{N}_{\ell m}^h = \ell(\ell+1)\,\mathcal{E}r_\ell^m + \frac{1}{r^2}\frac{\partial}{\partial r}\left[r^2\left\{(\ell+1)\,c_\ell^m\,\mathcal{E}t_{\ell-1}^m - \ell\,c_{\ell+1}^m\,\mathcal{E}t_{\ell+1}^m + im\,\mathcal{E}p_\ell^m\right\}\right]\ . \tag{4.40}$$

See also:

The θ and ϕ derivatives that enter (4.40) 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

$$\mathcal{C}_n(r)W_{\ell mn} = 0 \ \ \text{at} \ \ r = r_i, r_o\ . \tag{4.41}$$

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 when

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

and

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

for all spherical harmonic modes (ℓ, m). The conditions (4.41)-(4.43) 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 (4.43) 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 $\mathsf{S}r_{r\phi}$ and $\mathsf{S}_{r\theta}$ of the rate-of-strain tensor vanish. Translated to the spectral representation this requires

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

4.7.2 Magnetic boundary conditions and inner core

Magnetic boundary conditions at the interface with an insulating mantle or insulating inner core are similarly implemented. The toroidal magnetic field cannot enter any insulator and therefore has to vanish at the boundary

$$\mathcal{C}_n(r)h_{\ell mn} = 0 \ \ \text{at} \ \ r = r_i \ \ \text{and/or} \ \ r = r_o\ .$$

Matching conditions for the poloidal magnetic field with a source-free external potential field require that the following equations are satisfied at the boundary grid points:

$$\mathcal{C}'_n(r) g_{\ell mn} - \mathcal{C}_n(r) \frac{\ell+1}{r} g_{\ell mn} = 0 \quad \text{at} \quad r = r_i,$$

$$\mathcal{C}'_n(r) g_{\ell mn} + \mathcal{C}_n(r) \frac{\ell}{r} g_{\ell mn} = 0 \quad \text{at} \quad r = r_o.$$

If the inner core is modeled 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:

$$\int d\Omega \, Y_\ell^{m\star} \, \boldsymbol{e_r} \cdot \left[\boldsymbol{\nabla} \times \left(\boldsymbol{u^I} \times \boldsymbol{B^I} \right) \right] = -i \, \omega_I \, m \, \frac{\ell(\ell+1)}{r^2} \, g_{\ell m}^I(r) \,, \tag{4.44}$$

$$\int d\Omega \, Y_\ell^{m\star} \, \boldsymbol{e_r} \cdot \left[\boldsymbol{\nabla} \times \boldsymbol{\nabla} \times \left(\boldsymbol{u^I} \times \boldsymbol{B^I} \right) \right] = -i \, \omega_I \, m \, \frac{\ell(\ell+1)}{r^2} \, h_{\ell m}^I(r) \,. \tag{4.45}$$

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:

$$x(r) = \frac{r}{r_i} \,, \quad -r_i \leq r \leq r_i \,. \tag{4.46}$$

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:

$$g_{\ell m}^I(r) = \left(\frac{r}{r_i} \right)^{\ell+1} \sum_{i=0}^{M-1} g_{\ell m \, 2i}^I \, \mathcal{C}_{2i}(r) \,. \tag{4.47}$$

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 (4.44), (4.45) 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.

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)
     --no-recompile      Compile only once
     --hybrid            Run the hybrid version
```

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`)

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.

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 $t=tEND$. 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

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

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

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

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.

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

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

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

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

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: $t(r) \approx r \sin[\phi(r - r_o)]$. In case of a conducting inner core: $t(r) \approx r \sin[\pi(r/r_o)]$.

- `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: $b(r) \approx \left(\frac{r}{r_i}\right)^2 \left[1 - \frac{3}{5}\left(\frac{r}{r_o}\right)^2\right]$. 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 $r = r_i$: $t(r) \approx \left(\frac{r}{r_i}\right)^3 \left[1 - \left(\frac{r}{r_o}\right)^2\right]$.

- `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 ($\ell = 1, m = 0$): $t(r) = \text{ampb1} \left(\frac{r_i}{r}\right)^6$. 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=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

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

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

> **Warning:** The RMS calculation is actually wrong in the current version. This needs again to be ported from MagIC 3.44. The RMS contributions to the induction equation are correct, though. A ticket has been opened on github regarding this issue: https://github.com/magic-sph/magic/issues/1

The code can compute the RMS of the force balance and 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 the *dtVrms.TAG*, *dtBrms.TAG* and *dtDrms.TAG* files.

- **l_RMStest** (default `l_RMStest=.false.`) is a logical. This is a debug flag to check the consistency of the RMS calculation.

- **rCut** (default `rCut=0.075`) 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`.

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))
```

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
```

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*, *dtBrms.TAG* and *dtDrms.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

$$E_k = \frac{1}{2} \int_V \tilde{\rho} u^2 \, \mathrm{d}V = E_{pol} + E_{tor}$$

$$= \frac{1}{2} \sum_{\ell,m} \ell(\ell+1) \int_{r_i}^{r_o} \frac{1}{\tilde{\rho}} \left[\frac{\ell(\ell+1)}{r^2} |W_{\ell m}|^2 + \left| \frac{\mathrm{d}W_{\ell m}}{\mathrm{d}r} \right|^2 \right] \mathrm{d}r \qquad (8.1)$$

$$+ \frac{1}{2} \sum_{\ell,m} \ell(\ell+1) \int_{r_i}^{r_o} \frac{1}{\tilde{\rho}} |Z_{\ell m}|^2 \, \mathrm{d}r$$

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

$$
E_m = \frac{1}{2} \int_V B^2 \, \mathrm{d}V = E_{pol} + E_{tor}
$$

$$
= \frac{1}{2} \sum_{\ell,m} \ell(\ell+1) \int_{r_i}^{r_o} \left[\frac{\ell(\ell+1)}{r^2} |b_{\ell m}|^2 + \left| \frac{\mathrm{d}b_{\ell m}}{\mathrm{d}r} \right|^2 \right] \mathrm{d}r \tag{8.2}
$$

$$
+ \frac{1}{2} \sum_{\ell,m} \ell(\ell+1) \int_{r_i}^{r_o} |j_{\ell m}|^2 \, \mathrm{d}r
$$

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:

$$\mathcal{U} = \frac{1}{2} \int_V u^2 \, \mathrm{d}V = \mathcal{U}_{pol} + \mathcal{U}_{tor}$$

$$= \frac{1}{2} \sum_{\ell,m} \ell(\ell+1) \int_{r_i}^{r_o} \frac{1}{\tilde{\rho}^2} \left[\frac{\ell(\ell+1)}{r^2} |W_{\ell m}|^2 + \left| \frac{\mathrm{d}W_{\ell m}}{\mathrm{d}r} \right|^2 \right] \, \mathrm{d}r$$

$$+ \frac{1}{2} \sum_{\ell,m} \ell(\ell+1) \int_{r_i}^{r_o} \frac{1}{\tilde{\rho}^2} |Z_{\ell m}|^2 \, \mathrm{d}r$$

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`

> **Warning:** The RMS calculation is actually wrong in the current version. This needs again to be ported from MagIC 3.44. This issue only affects `dtVrms.TAG`, though. A ticket has been opened on github regarding this issue: https://github.com/magic-sph/magic/issues/1

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	Poloidal flow changes: inertia–advection
3	Toroidal flow changes: inertia–advection
4	Poloidal Coriolis force
5	Toroidal Coriolis force
6	Poloidal Lorentz force
7	Toroidal Lorentz force
8	Poloidal advection term
9	Toroidal advection term
10	Poloidal diffusion term
11	Toroidal diffusion term
12	Buoyancy term
13	Pressure gradient term
14	Sum of force terms: geostrophic balance
15	Sum of force terms: magnetostrophic balance
16	Sum of force terms: Archemidian balance

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 stretching term
5	Toroidal stretching term
6	Poloidal field advection term
7	Toroidal field advection term
8	Poloidal diffusion term
9	Toroidal diffusion term
10	Omega effect / toroidal stretching term
11	Omega effect
12	Poloidal field production (stretching+advection)
13	Toroidal field production (stretching+advection)

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 `dtDrms.TAG`

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

This files contains the RMS terms that enter the induction equation of the dipole. This file is written by the subroutine *dtBrms*.

No. of column	Contents
1	time
2	Dipole stretching
3	Dipole advection term
4	Dipole diffusion term

8.3.11 `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) or
                           !Graphout_version_7 (without MPI)

!----------
! 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
!-------------------

! This version is 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_mag):                                       !For a magnetic run

  !--------------
  ! Line 4 + (N+5)
  !--------------

  br(1:n_phi_tot, n_theta_start:n_theta_stop, n_r)  !Radial magnetic field

  !--------------
  ! Line 4 + (N+6)
  !--------------

  bt(1:n_phi_tot, n_theta_start:n_theta_stop, n_r)  !Theta component of
                                                    !magnetic field
```

```
   !--------------
   ! Line 4 + (N+7)
   !--------------

  bp(1:n_phi_tot, n_theta_start:n_theta_stop, n_r)  !Zonal (phi component)
                                                    !of magnetic field


!------------------
!Graphout_version_7
!------------------

!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_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 jth 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)
```

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. For additional diagnostics (boundary layer, heat transport, interpolation on cylindrical grids, etc.), see *here*.
12. 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)

__weakref__
list of weak references to the object (if defined)

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

> **__weakref__**
>>   list of weak references to the object (if defined)

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)

__weakref__
    list of weak references to the object (if defined)

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

__weakref__
list of weak references to the object (if defined)

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

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

> **`__weakref__`**
>
> list of weak references to the object (if defined)

> **`plot`**(*cut=0.8, levs=16, avg=True, cmap='RdYlBu_r'*)
>
> Plotting function

9.3. Python functions and classes

> **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

__weakref__
    list of weak references to the object (if defined)

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 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 <secBLayersRfile>* 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

__weakref__
    list of weak references to the object (if defined)

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

9.3.14 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*)

> 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
>
> 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*)

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

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

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** *blocking*, *dtb_mod* (*initialize_dtb_mod()*), *num_param*, *grenoble*, *outpv3* (*initialize_outpv3()*), *fields*, *precalculations*, *truncation*, *init_fields*, *timing* (*writetime()*, *walltime()*), *outto_mod* (*initialize_outto_mod()*), *namelists*, *kinetic_energy*, *rms* (*initialize_rms()*), *radialloop*, *fields_average_mod*, *spectra* (*initialize_spectra()*), *magnetic_energy*, *output_mod* (*initialize_output()*), *lmloop_mod* (*initialize_lmloop()*), *parallel_mod*, *matrices*, *outpar_mod*

> *(initialize_outpar_mod())*, *output_data*, *radial_functions*,
> *communications* *(initialize_communications())*, *logic*, *fieldslast*,
> *torsional_oscillations*, *start_fields* *(getstartfields())*,
> *step_time_mod* *(initialize_step_time()*, *step_time())*,
> *precision_mod*, *egeos_mod*, *physical_parameters*, *movie_data*
> *(initialize_movie_data()*, *finalize_movie_data())*, *radial_data*
> *(initialize_radial_data())*, *lmloop_data* *(initialize_lmloop_data())*,
> *horizontal_data*, *constants* *(codeversion())*, *power*
> *(initialize_output_power())*

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_rms()*,
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(), *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_out_real*, *sizeof_def_complex*, *sizeof_def_real*,
> *sizeof_character*, *cp*, *mpi_def_real*, *outp*, *lip*, *sizeof_integer*,
> *mpi_def_complex*, *mpi_out_real*

Needed modules

- `mpi`

Variables

- precision_mod/**sizeof_def_real** *[integer,parameter=8/public]*

- precision_mod/**lip** *[integer,parameter*=selected_int_kind(12)/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_complex** *[integer,parameter*=16/public*]*

- precision_mod/**mpi_out_real** *[integer,parameter*=mpi_real4/public*]*

- precision_mod/**sizeof_out_real** *[integer,parameter*=4/public*]*

- precision_mod/**cp** *[integer,parameter*=selected_real_kind(15)/public*]*

- precision_mod/**outp** *[integer,parameter*=selected_real_kind(6)/public*]*

- precision_mod/**sizeof_integer** *[integer,parameter*=4/public*]*

- precision_mod/**sizeof_character** *[integer,parameter*=1/public*]*

10.2.2 `truncation.f90`

Description

This module defines the grid points and the truncation

Quick access

Variables *l_maxmag*, *nrp*, *lm_maxmag*, *n_r_ic_maxmag*, *m_max*, *lgeos*, *nrpgeos*, *lmagmem*, *n_r_ic_max_ave*, *lmp_max_dtb*, *n_r_max_ave*, *lavemem*, *n_r_max_dtb*, *lm_max_real*, *n_phi_maxstr*, *ncp*, *n_r_totmag*, *minc*, *lmoviemem*, *n_r_maxgeos*, *n_phi_tot*, *lstressmem*, *lm_maxgeos*, *nalias*, *l_max*, *ncpgeos*, *lm_max_dtb*, *n_cheb_max*, *n_r_maxstr*, *n_m_max*, *n_r_ic_max*, *lm_max_ave*, *lmp_max*, *n_r_maxmag*, *n_r_ic_max_dtb*, *n_theta_maxstr*, *ldtbmem*, *n_r_max*, *n_r_tot*, *lm_max*, *n_cheb_ic_max*, *n_theta_max*, *n_phi_max*

Routines *initialize_truncation()*, *checktruncation()*

Variables

- truncation/**lstressmem** *[integer]*
      Memory for stress output

- truncation/**n_phi_tot** *[integer]*
      number of longitude grid points

- truncation/**n_r_max_ave** *[integer]*
      Number of radial points for time average

- truncation/**n_r_ic_maxmag** *[integer]*
      Number of radial points to calculate IC magnetic field

- truncation/**lavemem** *[integer]*
      Memory for calculating time averages

- truncation/**lm_max_ave** *[integer]*
      Number of l/m combinations for time average

- truncation/**lgeos** *[integer]*
    Memory for Geostrophic output

- truncation/**lm_max_real** *[integer]*
    number of l/m combination for real representation (cos/sin)

- truncation/**ncp** *[integer]*

- truncation/**n_r_maxstr** *[integer]*
    Number of radial points for stress output

- truncation/**n_r_maxmag** *[integer]*
    Number of radial points to calculate magnetic field

- truncation/**ldtbmem** *[integer]*
    Memory for movie output

- truncation/**n_r_ic_max_dtb** *[integer]*
    Number of IC radial points for movie output

- truncation/**lm_max** *[integer]*
    number of l/m combinations

- truncation/**nrp** *[integer]*
    dimension of phi points in for real/complex arrays

- truncation/**n_phi_max** *[integer]*
    absolute number of phi grid-points

- truncation/**n_r_max_dtb** *[integer]*
    Number of radial points for movie output

- truncation/**nrpgeos** *[integer]*
    Number of cyl. radial points for Geostrophic output

- truncation/**n_r_ic_max** *[integer]*
    number of grid points in inner core

- truncation/**lmp_max** *[integer]*
    number of l/m combination if l runs to l_max+1

- truncation/**n_r_max** *[integer]*
    number of radial grid points

- truncation/**n_r_totmag** *[integer]*
    n_r_maxMag + n_r_ic_maxMag

- truncation/**n_m_max** *[integer]*
    max number of ms (different oders)

- truncation/**l_max** *[integer]*
    max degree of Plms

- truncation/**lm_max_dtb** *[integer]*
    Number of l/m combinations for movie output

- truncation/**lmagmem** *[integer]*
    Memory for magnetic field calculation

- truncation/**l_maxmag** *[integer]*
    Max. degree for magnetic field calculation

- truncation/**ncpgeos** *[integer]*

- truncation/**n_theta_max** *[integer]*
    number of theta grid-points

- truncation/**n_theta_maxstr** *[integer]*
    Number of theta points for stress output

- truncation/**lm_maxmag** *[integer]*
    Max. number of l/m combinations for magnetic field calculation

- truncation/**n_phi_maxstr** *[integer]*
    Number of phi points for stress output

- truncation/**m_max** *[integer]*
    max order of Plms

- truncation/**lmp_max_dtb** *[integer]*
    Number of l/m combinations for movie output if l runs to l_max+1

- truncation/**n_cheb_max** *[integer]*
    max degree-1 of cheb polynomia

- truncation/**n_r_maxgeos** *[integer]*
    Number of radial points for Geostrophic output

- truncation/**n_r_ic_max_ave** *[integer]*
    Number of IC radial points for time average

- truncation/**nalias** *[integer]*
    controls dealiasing in latitude and

- truncation/**lmoviemem** *[integer]*
    Memory for movies

- truncation/**minc** *[integer]*
    basic wavenumber, longitude symmetry

- truncation/**n_r_tot** *[integer]*
    total number of radial grid points

- truncation/**n_cheb_ic_max** *[integer]*
    number of chebs in inner core

- truncation/**lm_maxgeos** *[integer]*
    Number of l/m combinations for Geostrophic output

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** *delxr2*, *delxh2*, *runtimestart*, *runtime*, *runtimelimit*, *vscale*, *alpha*,
> *dtmax*, *alffac*, *tscale*, *courfac*, *dtmin*, *enscale*, *intfac*, *dtstart*, *difkap*,
> *escale*, *tend*, *difnu*, *difeta*, *lscale*, *amstart*, *timestart*, *pscale*, *n_tscale*,
> *ldif*, *istop*, *n_lscale*, *ldifexp*, *n_time_steps*, *n_cour_step*

> **Routines** *initialize_num_param()*

Needed modules

- *truncation* (*n_r_max()*): This module defines the grid points and the truncation

- *precision_mod*: This module controls the precision used in MagIC

Variables

- num_param/**ldif** *[integer,public]*
      Degree where hyperdiffusion starts to act

- num_param/**tscale** *[real,public]*
      Time scale

- num_param/**dtmin** *[real,public]*
      Minimum allowed time step

- num_param/**intfac** *[real,public]*
      Value to re-scale dtMax during simulation

- num_param/**amstart** *[real,public]*

- num_param/**dtstart** *[real,public]*
      Initial time step if start solution is initialized

- num_param/**runtime** (4) *[integer,public]*
      Running time

- num_param/**delxr2** (:) *[real,allocatable/public]*

- num_param/**enscale** *[real,public]*
      Energies scale

- num_param/**pscale** *[real,public]*

- num_param/**alpha** *[real,public]*
      Weight for implicit time step

- num_param/**istop** *[integer,public]*
      Variable used in FFT soubroutine

- num_param/**difeta** *[real,public]*
      Amplitude of magnetic hyperdiffusion

- num_param/**n_time_steps** *[integer,public]*
    Total number of time steps requested in the name list

- num_param/**vscale** *[real,public]*
    Velocity scale

- num_param/**ldifexp** *[integer,public]*
    Exponent for hyperdiffusion function

- num_param/**lscale** *[real,public]*
    Length scale

- num_param/**timestart** *[real,public]*
    Numerical time where run should start

- num_param/**n_lscale** *[integer,public]*
    Control length scale

- num_param/**courfac** *[real,public]*
    Value to scale velocity in courant criteria

- num_param/**alffac** *[real,public]*
    Value to scale Alfen-velocity in courant criteria

- num_param/**runtimelimit** (4) *[integer,public]*
    Maximum running time

- num_param/**difnu** *[real,public]*
    Amplitude of viscous hyperdiffusion

- num_param/**dtmax** *[real,public]*
    Maximum allowed time step

- num_param/**delxh2** (:) *[real,allocatable/public]*

- num_param/**runtimestart** (4) *[integer,public]*
    Wall clock time of start of the run

- num_param/**tend** *[real,public]*
    Numerical time where run should end

- num_param/**escale** *[real,public]*
    Energy scale

- num_param/**n_cour_step** *[integer,public]*
    Step for controlling Courant criteria

- num_param/**difkap** *[real,public]*
    Amplitude of thermal hyperdiffusion

- num_param/**n_tscale** *[integer,public]*
    Control time 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** *n_imps_max*, *phis*, *peaks*, *thetas*, *widths*, *interior_model*, *polind*, *rho_ratio_ma*, *o_sr*, *pr*, *difexp*, *tmagcon*, *opr*, *r_cut_model*, *prmag*, *sigma_ratio*, *conductance_ma*, *epss*, *g0*, *con_lambdaout*, *rascaled*, *con_decrate*, *lffac*, *con_radratio*, *g2*, *opm*, *epsc*, *r_lcr*, *g1*, *strat*, *ohmlossfac*, *epsc0*, *ek*, *dissnb*, *slopestrat*, *vischeatfac*, *ekscaled*, *rho_ratio_ic*, *ra*, *buofac*, *cmbhflux*, *radratio*, *con_funcwidth*, *con_lambdamatch*, *corfac*, *mode*, *imagcon*, *n_r_lcr*, *nvarcond*, *ktops*, *imps*, *kbotb*, *kbotv*, *nvareps*, *ktopb*, *n_imps*, *nvarvisc*, *nvardiff*, *ktopv*, *kbots*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC

Variables

- physical_parameters/**nvareps** *[integer]*
    Selection of internal heating profile

- physical_parameters/**r_lcr** *[real]*
    Radius beyond which conductivity is zero

- physical_parameters/**ktopv** *[integer]*
    Velocity boundary condition

- physical_parameters/**epsc0** *[real]*
    Internal heat source magnitude

- physical_parameters/**rho_ratio_ma** *[real]*
    Same density as outer core

- physical_parameters/**tmagcon** *[real]*
    Time for magnetoconvection calculation

- physical_parameters/**kbotv** *[integer]*

- physical_parameters/**g1** *[real]*
    Set to 1.0 for linear gravity

- physical_parameters/**opr** *[real]*
    Inverse of Prandtl number

- physical_parameters/**nvarvisc** *[integer]*
    Selection of variable viscosity profile

- physical_parameters/**sigma_ratio** *[real]*
    Value of IC rotation

- physical_parameters/**imagcon** *[integer]*
    Imposed magnetic field for magnetoconvection, at the boundaries

- physical_parameters/**nvarcond** *[integer]*
    Selection of variable conductivity profile

- physical_parameters/**ek** *[real]*
    Ekman number

- physical_parameters/**opm** *[real]*
    Inverse of magnetic Prandtl number

- physical_parameters/**cmbhflux** *[real]*
    stratified Layer

- physical_parameters/**dissnb** *[real]*
    Dissipation number

- physical_parameters/**g2** *[real]*
    Set to 1.0 for $1/r^2$ gravity

- physical_parameters/**vischeatfac** *[real]*
    Prefactor for viscous heating: $Di\,Pr/Ra$

- physical_parameters/**o_sr** *[real]*
    Inverse of sigma_ratio

- physical_parameters/**g0** *[real]*
    Set to 1.0 for constant gravity

- physical_parameters/**nvardiff** *[integer]*
    Selection of variable diffusivity profile

- physical_parameters/**epss** *[real]*
    Deviation from the adiabat

- physical_parameters/**n_imps** *[integer]*
    Heat boundary condition

- physical_parameters/**lffac** *[real]*
    Inverse of Pr*Ekman

- physical_parameters/**difexp** *[real]*
    Thermal diffusivity variation

- physical_parameters/**strat** *[real]*
    number of density scale heights

- physical_parameters/**ktopb** *[integer]*
    Magnetic boundary condition

- physical_parameters/**r_cut_model** *[real]*
    Percentage on the inner part of the interior model to be used

- physical_parameters/**imps** *[integer]*
    Heat boundary condition

- physical_parameters/**con_funcwidth** *[real]*
    nVarCond=1

- physical_parameters/**pr** *[real]*
    Prandtl number

- physical_parameters/**con_lambdaout** *[real]*
    nVarCond=1

- physical_parameters/**conductance_ma** *[real]*
    OC conductivity

- physical_parameters/**ra** *[real]*
    Rayleigh number

- physical_parameters/**interior_model** *[character]*
    name of the interior model

- physical_parameters/**con_decrate** *[real]*
    Slope of electrical conductivity profile (nVarCond=2)

- physical_parameters/**buofac** *[real]*
    Ratio of Rayleigh number over Prandtl number

- physical_parameters/**radratio** *[real]*
    aspect ratio

- physical_parameters/**epsc** *[real]*
    Renormalisation of epsc0

- physical_parameters/**rascaled** *[real]*
    $Ra\, l^3$

- physical_parameters/**con_lambdamatch** *[real]*
    Electrical conductivity at con_RadRatio (nVarCond=2)

- physical_parameters/**thetas** (20) *[real]*

- physical_parameters/**prmag** *[real]*
    magnetic Prandtl number

- physical_parameters/**phis** (20) *[real]*

- physical_parameters/**kbots** *[integer]*

- physical_parameters/**n_imps_max** *[integer,parameter=20]*
    Heat boundary condition

- physical_parameters/**ohmlossfac** *[real]*
    Prefactor for Ohmic heating: $Di\, Pr/(Ra\, E\, Pm^2)$

- physical_parameters/**con_radratio** *[real]*
    Transition between branches of electrical conductivity profile (nVarCond=1,2)

- physical_parameters/**polind** *[real]*
    polytropic index

- physical_parameters/**kbotb** *[integer]*

- physical_parameters/**peaks** (20) *[real]*

- physical_parameters/**ekscaled** *[real]*
    $E\, l^2$

- physical_parameters/**widths** (20) *[real]*

- physical_parameters/**corfac** *[real]*
    Inverse of ekScaled

- physical_parameters/**slopestrat** *[real]*
    stratified Layer

- physical_parameters/**rho_ratio_ic** *[real]*
    Same density as outer core

- physical_parameters/**mode** *[integer]*
    Mode of calculation

- physical_parameters/**n_r_lcr** *[integer]*
    Number of radial points where conductivity is zero

- physical_parameters/**ktops** *[integer]*
    Entropy boundary condition

10.2.5 `logic.f90`

Description

Module containing the logicals that control the run

Quick access

> **Variables** *l_rmstest*, *l_cond_ma*, *l_anel*, *lverbose*, *l_heat_nl*, *l_anelastic_liquid*, *l_corr*, *l_tomovie*, *l_movie*, *l_rmagspec*, *l_htmovie*, *l_true_time*, *l_isothermal*, *l_runtimelimit*, *l_ht*, *l_power*, *l_corrmov*, *l_to*, *l_iner*, *l_dtbmovie*, *l_dtb*, *l_save_out*, *l_lcr*, *l_srma*, *l_storetpot*, *l_am*, *l_dt_cmb_field*, *l_fluxprofs*, *l_conv*, *l_pv*, *l_mag_lf*, *l_rms*, *l_cmb_field*, *l_storepot*, *l_r_field*, *l_conv_nl*, *l_viscbccalc*, *l_mag*, *l_par*, *l_storebpot*, *l_time_hits*, *l_movie_ic*, *l_update_v*, *l_dtrmagspec*, *l_rot_ma*, *l_b_nl_cmb*, *l_newmap*, *l_b_nl_icb*, *l_perppar*, *l_z10mat*, *l_average*, *l_update_s*, *l_movie_oc*, *l_heat*, *l_sric*, *l_r_fieldt*, *l_cond_ic*, *l_mag_kin*, *l_update_b*, *l_non_rot*, *l_store_frame*, *l_mag_nl*, *l_drift*, *l_rot_ic*, *l_storevpot*, *l_correct_ame*, *l_correct_amz*, *l_hel*

Variables

- logic/**l_rmstest** *[logical]*
    Switch for test of RMS balances (geostrophic, magnetostrophic, archimedean)

- logic/**l_dt_cmb_field** *[logical]*
    Switch for Bcoef files for secular variation of gauss coefs.

- logic/**l_cond_ma** *[logical]*
    Switch for conducting OC

- logic/**l_fluxprofs** *[logical]*
    Switch for calculation of radial profiles of flux contributions

- logic/**l_update_s** *[logical]*
    Switch off entropy update

- logic/**l_movie_oc** *[logical]*
    Switch for recording of movie files for OC

- logic/**l_corrmov** *[logical]*
    Switch for North/south correlation movie (see s_getEgeos.f)

- logic/**l_par** *[logical]*
    Switch for additional parameters calculation in s_getEgeos.f

- logic/**l_to** *[logical]*
    Switch for TO output in TOnhs.TAG, TOshs.TAG

- logic/**l_conv** *[logical]*
    Switch off convection

- logic/**l_mag_nl** *[logical]*
    Switch off non-linear magnetic terms calculation

- logic/**l_iner** *[logical]*
    Switch for inertial modes calculation

- logic/**l_mag_lf** *[logical]*
    Switch off Lorentz force term

- logic/**l_dtb** *[logical]*
    Switch to reserve memory for dtB movie

- logic/**l_pv** *[logical]*
    Switch for potential vorticity calculation

- logic/**l_movie** *[logical]*
    Switch for recording of movie files

- logic/**l_isothermal** *[logical]*
    Switch for isothermal calculation

- logic/**l_sric** *[logical]*
    Switch to rotating IC with prescribed rot. rate

- logic/**l_drift** *[logical]*
    Switch for drift rates calculation

- logic/**l_time_hits** *[logical]*
    Switch for time for outputs

- logic/**l_am** *[logical]*
    Switch for angular momentum calculation

- logic/**l_movie_ic** *[logical]*
    Switch for recording of movie files for IC

- logic/**l_srma** *[logical]*
    Switch to rotating OC with prescribed rot. rate

- logic/**l_heat** *[logical]*
    Switch off heat terms calculation

- logic/**l_rmagspec** *[logical]*
    Switch for magnetic spectra at different depths at log times

- logic/**l_r_fieldt** *[logical]*
    Switch for radial T coefficients

- logic/**l_anelastic_liquid** *[logical]*
    Switch for anelastic liquid calculation

- logic/**l_rms** *[logical]*
    Switch for RMS force balances calculation

- logic/**l_rot_ic** *[logical]*
    Switch off IC rotation

- logic/**l_update_v** *[logical]*
    Switch off velocity field update

- logic/**l_conv_nl** *[logical]*
    Switch off non-linear convection terms

- logic/**l_storebpot** *[logical]*
    Switch for storing magnetic field potentials

- logic/**l_non_rot** *[logical]*
    Switch to non-rotating

- logic/**l_storetpot** *[logical]*
    Switch for storing entropy field potentials

- logic/**l_tomovie** *[logical]*
    Switch for TO movie output

- logic/**l_true_time** *[logical]*
    Switch for times of outputs

- logic/**lverbose** *[logical]*
    Switch for detailed information about run progress

- logic/**l_heat_nl** *[logical]*
    Switch off non-linear heat terms calculation

- logic/**l_storevpot** *[logical]*
    Switch for storing velocity field potentials

- logic/**l_dtbmovie** *[logical]*
    Switch for dtB movie

- logic/**l_dtrmagspec** *[logical]*
    Switch for magnetic spectra at different depths at movie output times

- logic/**l_anel** *[logical]*
    Switch for anelastic calculation

- logic/**l_rot_ma** *[logical]*
    Switch off OC rotation

- logic/**l_viscbccalc** *[logical]*
    Switch for dissipation layer for stress-free BCs plots

- logic/**l_correct_ame** *[logical]*
    Switch for correction of equatorial angular mom.

- logic/**l_cmb_field** *[logical]*
    Switch for Bcoef files for gauss coefficients

- logic/**l_cond_ic** *[logical]*
    Switch for conducting IC

- logic/**l_newmap** *[logical]*
    Switch for non-linear mapping (see Bayliss and Turkel, 1990)

- logic/**l_htmovie** *[logical]*
    Switch for heat flux movie output

- logic/**l_save_out** *[logical]*
  Switch off outputs

- logic/**l_b_nl_icb** *[logical]*
  Switch for non-linear magnetic field at IC

- logic/**l_runtimelimit** *[logical]*
  Switch for absolute time limit of the run

- logic/**l_b_nl_cmb** *[logical]*
  Switch for non-linear magnetic field at OC

- logic/**l_mag_kin** *[logical]*
  Switch related for kinematic dynamo

- logic/**l_perppar** *[logical]*
  Switch for calculation of of kinetic energy perpendicular+parallel to the rotation axis

- logic/**l_correct_amz** *[logical]*
  Switch for correction of axial angular momentum

- logic/**l_update_b** *[logical]*
  Switch off magnetic field update

- logic/**l_z10mat** *[logical]*
  Switch for solid body rotation

- logic/**l_storepot** *[logical]*
  Switch for storing all field potentials

- logic/**l_ht** *[logical]*
  Switch for heat flux movie frame output

- logic/**l_lcr** *[logical]*
  Switch for zero electrical conductivity beyond r_LCR

- logic/**l_hel** *[logical]*
  Switch for helicity calculation, output in misc.TAG

- logic/**l_mag** *[logical]*
  Switch off magnetic terms calculation

- logic/**l_average** *[logical]*
  Switch for calculation of time-averages

- logic/**l_power** *[logical]*
  Switch for power budget terms calculation

- logic/**l_r_field** *[logical]*
  Switch for radial coefficients

- logic/**l_store_frame** *[logical]*
  Switch for storing movie frames

- 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** *ddw_lmloc*, *dp*, *b_ic_lmloc*, *ddj_ic_lmloc*, *b_rloc*, *ddb_lmloc*, *z_rloc*,
> *b_lmloc_container*, *b_ic*, *ddw*, *db*, *s*, *dj_lmloc*, *ddj_lmloc*, *db_lmloc*,
> *s_lmloc_container*, *z_lmloc*, *db_ic_lmloc*, *dp_lmloc*, *dj_rloc*, *ddw_rloc*,
> *aj_ic*, *dj*, *aj_rloc_container*, *aj_rloc*, *dz_lmloc*, *dz*, *aj_ic_lmloc*,
> *aj*, *z_lmloc_container*, *db_ic*, *dj_ic*, *ds*, *db_rloc*, *w_rloc*, *b_lmloc*,
> *ddb_rloc*, *aj_lmloc*, *w*, *dp_rloc*, *z_rloc_container*, *b*, *ddj_ic*, *dw_lmloc*,
> *ddb_ic_lmloc*, *s_lmloc*, *dz_rloc*, *s_rloc_container*, *ddj*, *p_rloc_container*,
> *p_lmloc*, *aj_lmloc_container*, *p*, *ds_rloc*, *dw*, *s_rloc*, *w_rloc_container*,
> *b_rloc_container*, *z*, *ds_lmloc*, *p_lmloc_container*, *w_lmloc*, *dj_ic_lmloc*,
> *ddb_ic*, *w_lmloc_container*, *ddb*, *dw_rloc*, *p_rloc*, *omega_ic*, *omega_ma*

> **Routines** *initialize_fields()*

Needed modules

- *truncation* (*n_r_ic_maxmag()*, *lm_max()*, *lm_maxmag()*, *n_r_maxmag()*, *n_r_max()*): This module defines the grid points and the truncation
- *lmloop_data* (*llm()*, *ulmmag()*, *llmmag()*, *ulm()*)
- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod* (rank()): This module contains the blocking information
- *radial_data* (*nrstart()*, *nrstop()*)

Variables

- fields/**ddw** (:,:) *[complex,allocatable/public]*
- fields/**dw_lmloc** (:,:) *[complex,pointer/public]*
- fields/**ddw_rloc** (:,:) *[complex,pointer/public]*
- fields/**dj_rloc** (:,:) *[complex,pointer/public]*
- fields/**b** (:,:) *[complex,allocatable/public]*
- fields/**z_rloc** (:,:) *[complex,pointer/public]*
- fields/**dp_lmloc** (:,:) *[complex,pointer/public]*
- fields/**b_lmloc_container** (:,:,:) *[complex,target/allocatable/public]*
- fields/**b_rloc_container** (:,:,:) *[complex,target/allocatable/public]*
- fields/**ddj_ic** (:,:) *[complex,allocatable/public]*
- fields/**ddw_lmloc** (:,:) *[complex,pointer/public]*
- fields/**dz_lmloc** (:,:) *[complex,pointer/public]*
- fields/**ddb_ic_lmloc** (:,:) *[complex,allocatable/public]*
- fields/**db_rloc** (:,:) *[complex,pointer/public]*
- fields/**dp** (:,:) *[complex,allocatable/public]*
- fields/**db** (:,:) *[complex,allocatable/public]*

- fields/**b_lmloc** (:,:) *[complex,pointer/public]*

- fields/**b_ic_lmloc** (:,:) *[complex,allocatable/public]*

- fields/**b_ic** (:,:) *[complex,allocatable/public]*

- fields/**z_lmloc** (:,:) *[complex,pointer/public]*

- fields/**ds** (:,:) *[complex,allocatable/public]*

- fields/**w** (:,:) *[complex,allocatable/public]*

- fields/**aj_lmloc_container** (:,:,:) *[complex,target/allocatable/public]*

- fields/**p** (:,:) *[complex,allocatable/public]*

- fields/**ddb_rloc** (:,:) *[complex,pointer/public]*

- fields/**p_lmloc_container** (:,:,:) *[complex,target/allocatable/public]*

- fields/**dz** (:,:) *[complex,allocatable/public]*

- fields/**ddj_ic_lmloc** (:,:) *[complex,allocatable/public]*

- fields/**s** (:,:) *[complex,allocatable/public]*

- fields/**dj_lmloc** (:,:) *[complex,pointer/public]*

- fields/**aj_ic_lmloc** (:,:) *[complex,allocatable/public]*

- fields/**dp_rloc** (:,:) *[complex,pointer/public]*

- fields/**aj** (:,:) *[complex,allocatable/public]*

- fields/**ds_rloc** (:,:) *[complex,pointer/public]*

- fields/**ddj_lmloc** (:,:) *[complex,pointer/public]*

- fields/**dw** (:,:) *[complex,allocatable/public]*

- fields/**ds_lmloc** (:,:) *[complex,pointer/public]*

- fields/**z** (:,:) *[complex,allocatable/public]*

- fields/**aj_ic** (:,:) *[complex,allocatable/public]*

- fields/**w_lmloc** (:,:) *[complex,pointer/public]*

- fields/**db_lmloc** (:,:) *[complex,pointer/public]*

- fields/**z_lmloc_container** (:,:,:) *[complex,target/allocatable/public]*

- fields/**aj_lmloc** (:,:) *[complex,pointer/public]*

- fields/**db_ic** (:,:) *[complex,allocatable/public]*

- fields/**dj_ic_lmloc** (:,:) *[complex,allocatable/public]*

- fields/**s_lmloc** (:,:) *[complex,pointer/public]*

- fields/**p_lmloc** (:,:) *[complex,pointer/public]*

- fields/**omega_ma** *[real,public]*

- fields/**w_lmloc_container** (:,:,:) *[complex,target/allocatable/public]*

- fields/**b_rloc** (:,:) *[complex,pointer/public]*

- fields/**dz_rloc** (:,:) *[complex,pointer/public]*

- fields/**s_lmloc_container** (:,:,:) *[complex,target/allocatable/public]*

- fields/**s_rloc** (:,:) *[complex,pointer/public]*

- fields/**w_rloc_container** (:,:,:) *[complex,target/allocatable/public]*

- fields/**dj** (:,:) *[complex,allocatable/public]*

- fields/**omega_ic** *[real,public]*

- fields/**s_rloc_container** (:,:,:) *[complex,target/allocatable/public]*

- fields/**ddb** (:,:) *[complex,allocatable/public]*

- fields/**ddj** (:,:) *[complex,allocatable/public]*

- fields/**ddb_lmloc** (:,:) *[complex,pointer/public]*

- fields/**ddb_ic** (:,:) *[complex,allocatable/public]*

- fields/**w_rloc** (:,:) *[complex,pointer/public]*

- fields/**db_ic_lmloc** (:,:) *[complex,allocatable/public]*

- fields/**aj_rloc_container** (:,:,:) *[complex,target/allocatable/public]*

- fields/**aj_rloc** (:,:) *[complex,pointer/public]*

- fields/**dw_rloc** (:,:) *[complex,pointer/public]*

- fields/**p_rloc_container** (:,:,:) *[complex,target/allocatable/public]*

- fields/**z_rloc_container** (:,:,:) *[complex,target/allocatable/public]*

- fields/**p_rloc** (:,:) *[complex,pointer/public]*

- fields/**dj_ic** (:,:) *[complex,allocatable/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** *dwdtlast_lmloc*, *djdtlast_lmloc*, *dwdtlast*, *djdt_iclast*,
> *djdt_iclast_lmloc*, *dpdtlast_lmloc*, *dpdtlast*, *dsdtlast*,
> *dbdt_iclast_lmloc*, *dbdt_iclast*, *dsdtlast_lmloc*, *djdtlast*, *dbdtlast*,
> *dbdtlast_lmloc*, *dzdtlast*, *dzdtlast_lo*, *lorentz_torque_iclast*,
> *d_omega_ic_dtlast*, *lorentz_torque_malast*, *d_omega_ma_dtlast*

> **Routines** *initialize_fieldslast()*

Needed modules

- *truncation* (*n_r_ic_maxmag()*, *n_r_max()*, *lm_maxmag()*, *n_r_maxmag()*, *lm_max()*): This module defines the grid points and the truncation
- *lmloop_data* (*llm()*, *ulmmag()*, *llmmag()*, *ulm()*)
- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod* (rank()): This module contains the blocking information

Variables

- fieldslast/**djdt_iclast** (:,:) *[complex,allocatable/public]*
- fieldslast/**dpdtlast** (:,:) *[complex,allocatable/public]*
- fieldslast/**dbdtlast** (:,:) *[complex,allocatable/public]*
- fieldslast/**lorentz_torque_malast** *[real,public]*
- fieldslast/**djdtlast** (:,:) *[complex,allocatable/public]*
- fieldslast/**dsdtlast** (:,:) *[complex,allocatable/public]*
- fieldslast/**dwdtlast** (:,:) *[complex,allocatable/public]*
- fieldslast/**dbdt_iclast_lmloc** (:,:) *[complex,allocatable/public]*
- fieldslast/**djdt_iclast_lmloc** (:,:) *[complex,allocatable/public]*
- fieldslast/**lorentz_torque_iclast** *[real,public]*
- fieldslast/**dzdtlast** (:,:) *[complex,allocatable/public]*
- fieldslast/**dwdtlast_lmloc** (:,:) *[complex,allocatable/public]*
- fieldslast/**dbdtlast_lmloc** (:,:) *[complex,allocatable/public]*
- fieldslast/**dzdtlast_lo** (:,:) *[complex,allocatable/public]*
- fieldslast/**dpdtlast_lmloc** (:,:) *[complex,allocatable/public]*
- fieldslast/**d_omega_ic_dtlast** *[real,public]*
- fieldslast/**dbdt_iclast** (:,:) *[complex,allocatable/public]*
- fieldslast/**d_omega_ma_dtlast** *[real,public]*
- fieldslast/**djdtlast_lmloc** (:,:) *[complex,allocatable/public]*
- fieldslast/**dsdtlast_lmloc** (:,:) *[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** *zmat*, *z10mat_fac*, *z10mat*, *jmat_fac*, *smat_fac*, *wpmat_fac*, *zmat_fac*, *s0mat*, *jmat*, *smat*, *bmat_fac*, *s0mat_fac*, *bmat*, *wpmat*, *s0pivot*, *lsmat*, *jpivot*, *lwpmat*, *lbmat*, *lzmat*, *bpivot*, *z10pivot*, *zpivot*, *spivot*, *wppivot*, *lz10mat*
>
> **Routines** *initialize_matrices()*

Needed modules

- *truncation* (*n_r_max()*, *l_max()*, *n_r_tot()*, *n_r_totmag()*, *l_maxmag()*): This module defines the grid points and the truncation
- *precision_mod*: This module controls the precision used in MagIC

Variables

- matrices/**z10mat** (:,:) *[real,allocatable/public]*
- matrices/**spivot** (:,:) *[integer,allocatable/public]*
- matrices/**wppivot** (:,:) *[integer,allocatable/public]*
- matrices/**smat_fac** (:,:) *[real,allocatable/public]*
- matrices/**smat** (:,:,:) *[real,allocatable/public]*
- matrices/**s0mat** (:,:) *[real,allocatable/public]*
- matrices/**jmat** (:,:,:) *[real,allocatable/public]*
- matrices/**lwpmat** (:) *[logical,allocatable/public]*
- matrices/**zmat_fac** (:,:) *[real,allocatable/public]*
- matrices/**zmat** (:,:,:) *[real,allocatable/public]*
- matrices/**lz10mat** *[logical,public]*
- matrices/**lsmat** (:) *[logical,allocatable/public]*
- matrices/**wpmat_fac** (:,:,:) *[real,allocatable/public]*
- matrices/**lzmat** (:) *[logical,allocatable/public]*
- matrices/**bmat** (:,:,:) *[real,allocatable/public]*
- matrices/**s0pivot** (:) *[integer,allocatable/public]*
- matrices/**wpmat** (:,:,:) *[real,allocatable/public]*
- matrices/**zpivot** (:,:) *[integer,allocatable/public]*
- matrices/**lbmat** (:) *[logical,allocatable/public]*

- matrices/**z10pivot** (:) *[integer,allocatable/public]*

- matrices/**z10mat_fac** (:) *[real,allocatable/public]*

- matrices/**jmat_fac** (:,:) *[real,allocatable/public]*

- matrices/**jpivot** (:,:) *[integer,allocatable/public]*

- matrices/**s0mat_fac** (:) *[real,allocatable/public]*

- matrices/**bmat_fac** (:,:) *[real,allocatable/public]*

- matrices/**bpivot** (:,:) *[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** *nzmaxa, nsmaxa, n_time_hits, t_r_file, v_r_file, b_r_file,*
> *cmbmov_file, e_mag_ic_file, u_square_file, cmb_file, e_kin_file,*
> *rst_file, e_mag_oc_file, dtvrms_file, log_file, dtvasrms_file, srma_file,*
> *par_file, dtbrms_file, perppar_file, dipole_file, dt_cmb_file, rot_file,*
> *power_file, angular_file, dtdrms_file, misc_file, lp_file, tag, runid,*
> *sric_file, graph_file, t_tomovie, t_bpot, t_vpot, t_pot, t_graph,*
> *t_toz, t_cmb, t_tpot, t_movie, t_rst, t_r_field, t_log, t_spec, t_to,*
> *n_coeff_r, n_b_r_file, n_t_r_file, n_r_array, n_v_r_file, t_bpot_stop,*
> *dt_to, dt_toz, t_rst_start, t_tpot_stop, dt_pot, t_cmb_stop, dt_movie,*
> *sdens, t_tomovie_start, dt_r_field, dt_cmb, t_movie_stop, rcut,*
> *t_toz_start, dt_graph, t_spec_start, dt_vpot, t_to_start, t_vpot_stop,*
> *t_graph_start, dt_bpot, dt_rst, dt_tpot, dt_tomovie, t_log_start,*
> *t_graph_stop, zdens, dt_log, t_toz_stop, t_to_stop, t_bpot_start,*
> *t_rst_stop, t_tomovie_stop, t_tpot_start, dt_spec, t_pot_start,*
> *t_cmb_start, t_r_field_start, t_vpot_start, t_movie_start, t_log_stop,*
> *t_spec_stop, rdea, t_pot_stop, t_r_field_stop, graph_mpi_fh, n_sric_file,*
> *rst_mpi_fh, n_e_mag_oc_file, n_srma_file, l_max_cmb, n_u_square_file,*
> *n_bpots, n_to_step, n_coeff_r_max, n_dtvasrms_file, n_t_to, n_t_bpot,*
> *l_graph_time, n_rot_file, n_tomovie_step, n_t_r_field, l_max_r, n_pots,*
> *n_t_cmb, n_mag_spec_file, n_cmbmov_file, n_power_file, n_t_graph,*
> *n_kin_spec_file, n_t_tomovie, n_bpot_step, n_stores, n_tos, n_misc_file,*
> *n_pot_step, n_tozs, nlf, n_r_field_step, n_log_file, n_t_toz, n_t_pot,*
> *n_movie_step, n_lp_file, n_r_step, n_angular_file, n_t_movie, n_graphs,*
> *n_dtdrms_file, n_vpots, n_dt_cmb_file, n_signal_file, n_rst_file,*
> *n_dtvrms_file, n_cmb_file, n_spec_step, n_t_tpot, n_tpot_step, n_tpots,*
> *n_perppar_file, n_dipole_file, n_specs, n_graph_step, n_toz_step,*

> > *n_tomovie_frames*, *n_t_log*, *n_rst_step*, *n_movie_frames*, *n_dtbrms_file*,
> > *n_t_spec*, *n_rsts*, *n_graph_file*, *n_r_fields*, *n_logs*, *n_par_file*,
> > *n_cmb_step*, *n_u2_spec_file*, *n_vpot_step*, *n_t_rst*, *n_log_step*, *n_t_vpot*,
> > *n_e_mag_ic_file*, *n_e_kin_file*, *n_cmbs*
>
> **Routines** *openfiles()*, *closefiles()*

Needed modules

- *charmanip* (*length_to_blank()*): This module contains several useful routines to manipule character strings

- *precision_mod*: This module controls the precision used in MagIC

- *logic* (*l_anel()*, *l_rot_ma()*, *l_am()*, *l_mag()*, *l_rmstest()*, *l_perppar()*, *l_power()*, *l_dt_cmb_field()*, *l_cmb_field()*, *l_r_fieldt()*, *l_r_field()*, *l_save_out()*, *l_rms()*, *l_rot_ic()*, *l_movie()*, *l_sric()*, *l_srma()*): Module containing the logicals that control the run

- *parallel_mod* (rank()): This module contains the blocking information

Variables

- output_data/**log_file** *[character,public]*

- output_data/**t_pot_stop** *[real,public]*

- output_data/**misc_file** *[character,public]*

- output_data/**t_spec_start** *[real,public]*

- output_data/**t_log_start** *[real,public]*

- output_data/**n_tpot_step** *[integer,public]*

- output_data/**t_spec** (5000) *[real,public]*

- output_data/**n_time_hits** *[integer,parameter=5000/public]*

- output_data/**n_t_toz** *[integer,public]*

- output_data/**t_bpot_start** *[real,public]*

- output_data/**n_srma_file** *[integer,public]*

- output_data/**n_toz_step** *[integer,public]*

- output_data/**n_e_mag_ic_file** *[integer,public]*

- output_data/**t_vpot_start** *[real,public]*

- output_data/**t_cmb_stop** *[real,public]*

- output_data/**dt_cmb_file** *[character,public]*

- output_data/**rot_file** *[character,public]*

- output_data/**t_r_field** (5000) *[real,public]*

- output_data/**n_t_tomovie** *[integer,public]*

- output_data/**n_sric_file** *[integer,public]*

- output_data/**rst_mpi_fh** *[integer,public]*

- output_data/**t_bpot_stop** *[real,public]*

- output_data/**cmbmov_file** *[character,public]*

- output_data/**t_cmb_start** *[real,public]*

- output_data/**t_to_stop** *[real,public]*

- output_data/**t_rst** (5000) *[real,public]*

- output_data/**dt_movie** *[real,public]*

- output_data/**n_e_mag_oc_file** *[integer,public]*

- output_data/**dt_log** *[real,public]*

- output_data/**n_t_log** *[integer,public]*

- output_data/**nzmaxa** *[integer,parameter*=194/public*]*

- output_data/**n_tos** *[integer,public]*

- output_data/**n_mag_spec_file** *[integer,public]*

- output_data/**t_log_stop** *[real,public]*

- output_data/**dt_vpot** *[real,public]*

- output_data/**t_vpot_stop** *[real,public]*

- output_data/**t_tomovie_stop** *[real,public]*

- output_data/**t_tomovie** (5000) *[real,public]*

- output_data/**runid** *[character,public]*

- output_data/**rdea** *[real,public]*

- output_data/**t_bpot** (5000) *[real,public]*

- output_data/**n_movie_step** *[integer,public]*

- output_data/**dt_graph** *[real,public]*

- output_data/**n_tozs** *[integer,public]*

- output_data/**n_rst_step** *[integer,public]*

- output_data/**n_t_spec** *[integer,public]*

- output_data/**n_cmbmov_file** *[integer,public]*

- output_data/**l_max_cmb** *[integer,public]*

- output_data/**n_power_file** *[integer,public]*

- output_data/**n_par_file** *[integer,public]*

- output_data/**dt_to** *[real,public]*

- output_data/**t_rst_start** *[real,public]*

- output_data/**t_movie** (5000) *[real,public]*

- output_data/**t_graph_start** *[real,public]*

- output_data/**n_lp_file** *[integer,public]*

- output_data/**n_v_r_file** (:) *[integer,allocatable/public]*

- output_data/**n_u_square_file** *[integer,public]*

- output_data/**t_rst_stop** *[real,public]*

- output_data/**t_toz_stop** *[real,public]*

- output_data/**n_b_r_file** (:) *[integer,allocatable/public]*

- output_data/**dt_bpot** *[real,public]*

- output_data/**t_tpot_stop** *[real,public]*

- output_data/**dt_pot** *[real,public]*

- output_data/**graph_mpi_fh** *[integer,public]*

- output_data/**dt_rst** *[real,public]*

- output_data/**e_mag_ic_file** *[character,public]*

- output_data/**n_signal_file** *[integer,public]*

- output_data/**n_spec_step** *[integer,public]*

- output_data/**n_t_r_file** (:) *[integer,allocatable/public]*

- output_data/**t_r_file** (:) *[character,allocatable/public]*

- output_data/**n_t_pot** *[integer,public]*

- output_data/**n_bpots** *[integer,public]*

- output_data/**n_rst_file** *[integer,public]*

- output_data/**t_pot** (5000) *[real,public]*

- output_data/**zdens** *[real,public]*

- output_data/**u_square_file** *[character,public]*

- output_data/**dt_toz** *[real,public]*

- output_data/**srma_file** *[character,public]*

- output_data/**dt_tpot** *[real,public]*

- output_data/**n_coeff_r_max** *[integer,public]*

- output_data/**power_file** *[character,public]*

- output_data/**n_log_step** *[integer,public]*

- output_data/**n_movie_frames** *[integer,public]*

- output_data/**dt_tomovie** *[real,public]*

- output_data/**n_r_step** *[integer,public]*

- output_data/**cmb_file** *[character,public]*

- output_data/**par_file** *[character,public]*

- output_data/**n_dtbrms_file** *[integer,public]*

- output_data/**n_stores** *[integer,public]*

- output_data/**n_rsts** *[integer,public]*

- output_data/**n_graphs** *[integer,public]*

- output_data/**n_dtvasrms_file** *[integer,public]*

- output_data/**e_mag_oc_file** *[character,public]*

- output_data/**n_r_fields** *[integer,public]*
- output_data/**n_t_rst** *[integer,public]*
- output_data/**n_t_movie** *[integer,public]*
- output_data/**n_t_to** *[integer,public]*
- output_data/**tag** *[character,public]*
- output_data/**n_r_array** (100) *[integer,public]*
- output_data/**t_to_start** *[real,public]*
- output_data/**t_spec_stop** *[real,public]*
- output_data/**t_cmb** (5000) *[real,public]*
- output_data/**dtbrms_file** *[character,public]*
- output_data/**t_graph_stop** *[real,public]*
- output_data/**n_pot_step** *[integer,public]*
- output_data/**n_bpot_step** *[integer,public]*
- output_data/**perppar_file** *[character,public]*
- output_data/**n_pots** *[integer,public]*
- output_data/**sric_file** *[character,public]*
- output_data/**sdens** *[real,public]*
- output_data/**n_dtdrms_file** *[integer,public]*
- output_data/**n_logs** *[integer,public]*
- output_data/**t_movie_start** *[real,public]*
- output_data/**n_vpots** *[integer,public]*
- output_data/**t_tpot_start** *[real,public]*
- output_data/**n_dt_cmb_file** *[integer,public]*
- output_data/**e_kin_file** *[character,public]*
- output_data/**n_cmb_step** *[integer,public]*
- output_data/**n_t_bpot** *[integer,public]*
- output_data/**t_tomovie_start** *[real,public]*
- output_data/**nlf** *[integer,public]*
- output_data/**dt_spec** *[real,public]*
- output_data/**n_t_tpot** *[integer,public]*
- output_data/**n_tomovie_frames** *[integer,public]*
- output_data/**dipole_file** *[character,public]*
- output_data/**t_graph** (5000) *[real,public]*
- output_data/**n_dtvrms_file** *[integer,public]*
- output_data/**n_u2_spec_file** *[integer,public]*
- output_data/**t_toz** (5000) *[real,public]*

- output_data/**n_to_step** *[integer,public]*
- output_data/**n_vpot_step** *[integer,public]*
- output_data/**l_graph_time** *[logical,public]*
- output_data/**n_cmb_file** *[integer,public]*
- output_data/**n_t_graph** *[integer,public]*
- output_data/**n_graph_step** *[integer,public]*
- output_data/**n_graph_file** *[integer,public]*
- output_data/**n_log_file** *[integer,public]*
- output_data/**n_angular_file** *[integer,public]*
- output_data/**dt_r_field** *[real,public]*
- output_data/**dt_cmb** *[real,public]*
- output_data/**n_t_vpot** *[integer,public]*
- output_data/**lp_file** *[character,public]*
- output_data/**angular_file** *[character,public]*
- output_data/**n_dipole_file** *[integer,public]*
- output_data/**n_rot_file** *[integer,public]*
- output_data/**nsmaxa** *[integer,parameter=97/public]*
- output_data/**n_t_cmb** *[integer,public]*
- output_data/**t_pot_start** *[real,public]*
- output_data/**rst_file** *[character,public]*
- output_data/**dtvasrms_file** *[character,public]*
- output_data/**dtdrms_file** *[character,public]*
- output_data/**t_to** (5000) *[real,public]*
- output_data/**v_r_file** (:) *[character,allocatable/public]*
- output_data/**n_kin_spec_file** *[integer,public]*
- output_data/**b_r_file** (:) *[character,allocatable/public]*
- output_data/**n_tomovie_step** *[integer,public]*
- output_data/**t_movie_stop** *[real,public]*
- output_data/**dtvrms_file** *[character,public]*
- output_data/**n_perppar_file** *[integer,public]*
- output_data/**rcut** *[real,public]*
- output_data/**n_misc_file** *[integer,public]*
- output_data/**t_toz_start** *[real,public]*
- output_data/**t_log** (5000) *[real,public]*
- output_data/**n_t_r_field** *[integer,public]*
- output_data/**n_specs** *[integer,public]*

- output_data/**t_r_field_stop** *[real,public]*

- output_data/**l_max_r** *[integer,public]*

- output_data/**n_tpots** *[integer,public]*

- output_data/**graph_file** *[character,public]*

- output_data/**n_coeff_r** (:) *[integer,allocatable/public]*

- output_data/**t_vpot** (5000) *[real,public]*

- output_data/**n_r_field_step** *[integer,public]*

- output_data/**t_r_field_start** *[real,public]*

- output_data/**n_e_kin_file** *[integer,public]*

- output_data/**n_cmbs** *[integer,public]*

- output_data/**t_tpot** (5000) *[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 *fac_loop*, *expo_imp*, *amp_curr*, *amp_imp*, *rrmp*, *bmax_imp*, *n_imp*, *l_imp*, *l_curr*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC

Variables

- bext/**l_imp** *[integer]*
    Mode of external field (dipole,quadrupole etc.)

- bext/**bmax_imp** *[real]*
    Location of maximum in g_ext/g_int

- bext/**rrmp** *[real]*
    Magnetopause radius

- bext/**fac_loop** (:) *[real,allocatable]*

- bext/**n_imp** *[integer]*
    Controls external field model

- bext/**amp_imp** *[real]*
    Amplitude of the time varying osc

- bext/**expo_imp** *[real]*
    Exponent for decay

- bext/**l_curr** *[logical]*
    Switch for current loop at the equator

- bext/**amp_curr** *[real]*
    Amplitude of magnetic field of current loop

10.2.11 `constants.f90`

Description

module containing constants and parameters used in the code.

Quick access

> **Variables** *codeversion*, *sin72*, *cos72*, *sq4pi*, *osq4pi*, *cos36*, *third*, *sin36*, *pi*, *zero*,
> *three*, *ci*, *sin60*, *two*, *half*, *four*, *one*, *vol_ic*, *c_dt_z10_ic*, *c_z10_omega_ma*,
> *mass*, *c_moi_ma*, *c_z10_omega_ic*, *c_dt_z10_ma*, *y10_norm*, *c_moi_oc*, *vol_oc*,
> *c_moi_ic*, *c_lorentz_ma*, *c_lorentz_ic*, *surf_cmb*, *y11_norm*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC

Variables

- constants/**c_moi_ma** *[real]*
    Moment of inertia of the mantle

- constants/**vol_ic** *[real]*
    Volume of the inner core

- constants/**cos36** *[real,parameter=*cos(36.0_cp*pi/180.0_cp)*]*

- `constants/`**sin72** *[real,parameter=*sin(72.0_cp**pi*/180.0_cp)*]*

- `constants/`**c_lorentz_ic** *[real]*

- `constants/`**c_z10_omega_ic** *[real]*

- `constants/`**c_moi_oc** *[real]*
      Moment of inertia of the outer core

- `constants/`**sin60** *[real,parameter=*0.5_cp*sqrt(3.0_cp)*]*

- `constants/`**one** *[real,parameter=*1.0_cp*]*

- `constants/`**half** *[real,parameter=*0.5_cp*]*

- `constants/`**zero** *[complex,parameter=(0.0_cp,0.0_cp)]*

- `constants/`**four** *[real,parameter=*4.0_cp*]*

- `constants/`**three** *[real,parameter=*3.0_cp*]*

- `constants/`**sin36** *[real,parameter=*sin(36.0_cp**pi*/180.0_cp)*]*

- `constants/`**surf_cmb** *[real]*
      Outer boundary surface

- `constants/`**c_dt_z10_ma** *[real]*

- `constants/`**y11_norm** *[real]*

- `constants/`**y10_norm** *[real]*

- `constants/`**vol_oc** *[real]*
      Volume of the outer core

- `constants/`**pi** *[real,parameter=*4.0_cp*atan(1.0_cp)*]*

- `constants/`**c_z10_omega_ma** *[real]*

- `constants/`**osq4pi** *[real,parameter=*1.0_cp/*sq4pi*]*

- `constants/`**cos72** *[real,parameter=*cos(72.0_cp**pi*/180.0_cp)*]*

- `constants/`**codeversion** *[character,parameter=*'5.2'*]*

- `constants/`**ci** *[complex,parameter=(0.0_cp,1.0_cp)]*

- `constants/`**sq4pi** *[real,parameter=*sqrt(4.0_cp**pi*)*]*

- `constants/`**two** *[real,parameter=*2.0_cp*]*

- `constants/`**mass** *[real]*
      Mass of the outer core

- `constants/`**c_lorentz_ma** *[real]*

- `constants/`**third** *[real,parameter=*one*/*three*]*

- `constants/`**c_dt_z10_ic** *[real]*

- `constants/`**c_moi_ic** *[real]*
      Moment of inertia of the inner core

10.2.12 `Grenoble.f90`

Description

This module contains all variables for the case of an imposed IC dipole

Quick access

> **Variables** *b0*, *db0*, *ddb0*, *bic*, *lgrenoble*
>
> **Routines** *initialize_grenoble()*

Needed modules

- *truncation* (*n_r_maxmag()*): This module defines the grid points and the truncation
- *precision_mod*: This module controls the precision used in MagIC

Variables

- grenoble/**lgrenoble** *[logical,public]*
- grenoble/**bic** *[real,public]*
- grenoble/**db0** (:) *[real,allocatable/public]*
- grenoble/**b0** (:) *[real,allocatable/public]*
- grenoble/**ddb0** (:) *[real,allocatable/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** *n_procs*, *nr_on_last_rank*, *rank_with_l1m0*, *nthreads*, *rank_bn*, *nr_per_rank*, *ierr*, *nlmbs_per_rank*, *chunksize*
>
> **Routines** *parallel()*

Needed modules

- `mpi`

- `omp_lib`

Variables

- `parallel_mod`/**`nlmbs_per_rank`** *[integer]*

- `parallel_mod`/**`nthreads`** *[integer]*

- `parallel_mod`/**`rank_bn`** *[integer]*

- `parallel_mod`/**`rank_with_l1m0`** *[integer]*

- `parallel_mod`/**`n_procs`** *[integer]*

- `parallel_mod`/**`nr_on_last_rank`** *[integer]*

- `parallel_mod`/**`nr_per_rank`** *[integer]*

- `parallel_mod`/**`chunksize`** *[integer]*

- `parallel_mod`/**`ierr`** *[integer]*

Subroutines and functions

subroutine `parallel_mod`/**`parallel`**()

> **Called from** *magic*

10.3.2 `radial_data.f90`

Quick access

> **Variables** *n_r_icb*, *nrstartmag*, *nrstart*, *nrstopmag*, *nrstop*, *n_r_cmb*
>
> **Routines** *initialize_radial_data()*

Needed modules

- *truncation* (*n_r_max()*): This module defines the grid points and the truncation

- *logic* (*l_mag()*, *lverbose()*): Module containing the logicals that control the run

- *parallel_mod* (*n_procs()*, *nr_on_last_rank()*, rank(), *nr_per_rank()*): This module contains the blocking information

Variables

- `radial_data`/**`nrstartmag`** *[integer,public]*

- `radial_data`/**`nrstopmag`** *[integer,public]*

- `radial_data`/**`n_r_cmb`** *[integer,public]*

- radial_data/**n_r_icb** *[integer,public]*
- radial_data/**nrstart** *[integer,public]*
- radial_data/**nrstop** *[integer,public]*

Subroutines and functions

subroutine radial_data/**initialize_radial_data**()

> **Called from** *magic*

10.3.3 `LMLoop_data.f90`

Quick access

> **Variables** *ulmmag*, *llm*, *lm_per_rank*, *ulm*, *llmmag*, *lm_on_last_rank*
>
> **Routines** *initialize_lmloop_data()*

Needed modules

- *blocking* (*nlmbs()*, *lmstartb()*, *lmstopb()*, *sizelmb()*): Module containing blocking information
- *logic* (*l_mag()*): Module containing the logicals that control the run
- *parallel_mod* (*nlmbs_per_rank()*, *n_procs()*, rank()): This module contains the blocking information

Variables

- lmloop_data/**llm** *[integer,public]*
- lmloop_data/**llmmag** *[integer,public]*
- lmloop_data/**lm_per_rank** *[integer,public]*
- lmloop_data/**ulm** *[integer,public]*
- lmloop_data/**lm_on_last_rank** *[integer,public]*
- lmloop_data/**ulmmag** *[integer,public]*

Subroutines and functions

subroutine lmloop_data/**initialize_lmloop_data**()

> **Called from** *magic*

10.3.4 `communications.f90`

Quick access

Variables *temp_r2lo*, *temp_gather_lo*, *s_transfer_type_nr_end*, *r_transfer_type_nr_end_cont*, *s_transfer_type*, *array_of_statuses*, *s_transfer_type_cont*, *r_request*, *r_transfer_type_cont*, *final_wait_array*, *r_transfer_type_nr_end*, *r_transfer_type*, *s_request*, *s_transfer_type_nr_end_cont*, *get_global_sum*, *gt_cheb*, *lo2r_p*, *lo2r_w*, *gt_oc*, *lo2r_aj*, *lo2r_s*, *lo2r_b*, *gt_ic*, *lo2r_z*, *r_lm_gather_type*, *r_lm_gather_type_lm_end*

Routines *myallgather()*, *get_global_sum_cmplx_2d()*, *get_global_sum_real_2d()*, *get_global_sum_cmplx_1d()*, *scatter_from_rank0_to_lo()*, *destroy_gather_type()*, *destroy_lm2r_type()*, *lo2r_redist_start()*, *lm2r_redist_start()*, *r2lm_redist()*, *lo2lm_redist()*, *gather_from_lo_to_rank0()*, *create_lm2r_type()*, *r2lo_redist()*, *lm2lo_redist()*, *initialize_communications()*, *create_gather_type()*, *gather_all_from_lo_to_rank0()*, *lm2r_redist_wait()*, *lo2r_redist_wait()*

Needed modules

- *blocking* (*lmstartb()*, *lmstopb()*, *st_map()*, *lo_map()*): Module containing blocking information

- *precision_mod*: This module controls the precision used in MagIC

- *radial_data* (*nrstart()*, *nrstop()*)

- *parallel_mod* (*n_procs()*, *nr_on_last_rank()*, *ierr()*, rank(), *nr_per_rank()*): This module contains the blocking information

- *logic* (*l_heat()*, *l_mag()*, *l_conv()*): Module containing the logicals that control the run

- *truncation* (*lm_max()*, *l_max()*, *n_r_ic_max()*, *n_r_max()*, *minc()*): This module defines the grid points and the truncation

- *lmloop_data* (*llm()*, *ulm()*)

- mpi

Types

- **type** communications/**unknown_type**

    Type fields

    – % **final_wait_array** (*) *[integer,allocatable]*

    – % **arr_rloc** („*) *[complex,pointer]*

    – % **temp_rloc** („*) *[complex,pointer]*

    – % **r_request** (*) *[integer,allocatable]*

    – % **count** *[integer]*

    – % **s_request** (*) *[integer,allocatable]*

- **type** communications/**unknown_type**

    Type fields

    - **% gather_mpi_type** (*) *[integer,allocatable]*
    - **% dim2** *[integer]*

Variables

- communications/**r_transfer_type_nr_end_cont** (:,:) *[integer,private/allocatable/save]*
- communications/**r_transfer_type** (:) *[integer,private/allocatable/save]*
- communications/**gt_cheb** *[gather_type,public]*
- communications/**lo2r_w** *[lm2r_type,public]*
- communications/**s_request** (:) *[integer,private/allocatable]*
- communications/**r_lm_gather_type** *[integer,private]*
- communications/**r_transfer_type_cont** (:,:) *[integer,private/allocatable/save]*
- communications/**s_transfer_type_nr_end_cont** (:,:) *[integer,private/allocatable/save]*
- communications/**r_request** (:) *[integer,private/allocatable]*
- communications/**temp_gather_lo** (:) *[complex,private/allocatable]*
- communications/**array_of_statuses** (:,:) *[integer,private/allocatable]*
- communications/**lo2r_p** *[lm2r_type,public]*
- communications/**r_transfer_type_nr_end** (:) *[integer,private/allocatable/save]*
- communications/**s_transfer_type_cont** (:,:) *[integer,private/allocatable/save]*
- communications/**get_global_sum** *[public]*
- communications/**gt_ic** *[gather_type,public]*
- communications/**temp_r2lo** (:,:) *[complex,private/allocatable]*
- communications/**lo2r_z** *[lm2r_type,public]*
- communications/**s_transfer_type** (:) *[integer,private/allocatable/save]*
- communications/**lo2r_s** *[lm2r_type,public]*
- communications/**r_lm_gather_type_lm_end** *[integer,private]*
- communications/**lo2r_b** *[lm2r_type,public]*
- communications/**final_wait_array** (:) *[integer,private/allocatable]*
- communications/**gt_oc** *[gather_type,public]*
- communications/**lo2r_aj** *[lm2r_type,public]*
- communications/**s_transfer_type_nr_end** (:) *[integer,private/allocatable/save]*

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** *outto()*, *output()*, *outpv()*, *getegeos()*, *fields_average()*, *get_dtblmfinish()*

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 *storepotw()*, *fields_average()*

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 *getstartfields()*, *lmloop()*

    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 *blocking*, *parallel_mod*

Called from *dtvrms()*, *dtbrms()*

10.4 Code initialization

10.4.1 `Namelists.f90`

Description

Read and print the input namelist

Quick access

>   **Routines** *defaultnamelists()*, *writenamelists()*, *readnamelists()*

Needed modules

- *bext*: Module containing the external field parameters
- *blocking* (*cacheblock_size_in_b()*): Module containing blocking information
- *num_param*: Module containing numerical and control parameters
- *grenoble*: This module contains all variables for the case of an imposed IC dipole
- *parallel_mod*: This module contains the blocking information
- *output_data*: This module contains the parameters for output control
- *radial_functions*: This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *logic*: Module containing the logicals that control the run
- *init_fields*
- *torsional_oscillations*: This module contains information for TO calculation and output
- *truncation*: This module defines the grid points and the truncation
- *precision_mod*: This module controls the precision used in MagIC
- *physical_parameters*: Module containing the physical parameters
- *movie_data* (*movie()*, *n_movies()*, *n_movies_max()*)
- *charmanip* (*capitalize()*, *length_to_blank()*): This module contains several useful routines to manipule character strings
- *constants*: module containing constants and parameters used in the code.

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

- *blocking* (*nlmbs()*, *lmstartb()*, *lmstopb()*, *lo_map()*): Module containing blocking information
- *communications* (*scatter_from_rank0_to_lo()*, *lo2r_b()*, *lo2r_redist_start()*, *get_global_sum()*, *lo2r_p()*, *lo2r_aj()*, *lo2r_w()*, *lo2r_s()*, *lo2r_z()*)
- *num_param* (*dtmax()*, *alpha()*): Module containing numerical and control parameters
- *grenoble* (*lgrenoble()*): This module contains all variables for the case of an imposed IC dipole
- mpi
- *fields*: This module contains the potential fields and their radial derivatives
- *radial_data* (*n_r_cmb()*, *n_r_icb()*)
- *truncation*: This module defines the grid points and the truncation
- *radial_functions* (*botcond()*, *drx()*, *topcond()*, *d_costf1_ic_init()*, *i_costf2_ic_init()*, *d_costf2_ic_init()*, *or1()*, *ddrx()*, *dtemp0()*, *i_costf_init()*, *r()*, *d_costf_init()*, *i_costf1_ic_init()*, *dr_fac_ic()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *logic* (*l_lcr()*, *l_rot_ma()*, *l_mag()*, *l_mag_kin()*, *l_conv()*, *l_sric()*, *l_cond_ic()*, *l_mag_lf()*, *l_z10mat()*, *l_rot_ic()*, *l_heat()*, *l_srma()*): Module containing the logicals that control the run
- *init_fields* (*l_start_file()*, *s_cond()*, *initb()*, *initv()*, *tops()*, *init_s1()*, *inits()*, *start_file()*, *init_b1()*)
- *useful* (*logwrite()*, *cc2real()*): 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.
- *radial_der* (*get_ddr()*, *get_dr()*): Radial derivatives functions
- *precision_mod*: This module controls the precision used in MagIC
- *radial_der_even* (*get_ddr_even()*)
- *physical_parameters* (*epss()*, *ktopv()*, *lffac()*, *imps()*, *imagcon()*, *interior_model()*, *kbotv()*, *n_r_lcr()*): Module containing the physical parameters
- *lmloop_data* (*llm()*, *ulmmag()*, *llmmag()*, *lm_per_rank()*, *ulm()*, *lm_on_last_rank()*)

- *parallel_mod* (*n_procs()*, *nlmbs_per_rank()*, rank()): This module contains the blocking information

- *readcheckpoints* (*readstartfields()*): This module contains the functions that can help reading and mapping of the restart files

- *constants* (*c_lorentz_ic()*, *c_lorentz_ma()*, *one()*, *two()*, *osq4pi()*, *zero()*): module containing constants and parameters used in the code.

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** *n_s_bounds*, *start_file*, *bots*, *s_top*, *s_bot*, *tops*, *tomega_ma2*, *omega_ic2*, *omegaosz_ma1*, *amp_s2*, *tshift_ma1*, *scale_s*, *omega_ma2*, *amp_v1*, *omega_ma1*, *amp_b1*, *omegaosz_ic2*, *tomega_ic2*, *tomega_ic1*, *tshift_ic2*, *omegaosz_ic1*, *tipdipole*, *omega_ic1*, *scale_v*, *tshift_ic1*, *bpeaktop*, *omegaosz_ma2*, *amp_s1*, *tshift_ma2*, *scale_b*, *tomega_ma1*, *bpeakbot*, *inform*, *init_s2*, *l_start_file*, *init_v1*, *nrotma*, *n_start_file*, *nrotic*, *l_reset_t*, *init_b1*, *init_s1*

> **Routines** *inits()*, *s_cond()*, *initv()*, *initialize_init_fields()*, *j_cond()*, *initb()*

Needed modules

- *blocking* (*nfs()*, *sizethetab()*, *lmp2lmps()*, *nthetabs()*, *st_map()*): Module containing blocking information

- *algebra* (*cgesl()*, *sgefa()*, *sgesl()*)

- *horizontal_data* (*d_lp1()*, *hdif_b()*, *dlh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

---

- *matrices* (*jpivot()*, *s0pivot()*, *s0mat()*, *jmat()*): This module contains matricies for internal time step

- *radial_data* (*n_r_cmb()*, *n_r_icb()*)

- *truncation*: This module defines the grid points and the truncation

- *radial_functions* (*cheb_ic()*, *dcheb_ic()*, *cheb()*, *d2cheb_ic()*, *or1()*, *kappa()*, *dlkappa()*, *jvarcon()*, *epscprof()*, *beta()*, *orho1()*, *cheb_norm()*, *i_costf_init()*, *or2()*, *i_costf1_ic_init()*, *dllambda()*, *d_costf1_ic_init()*, *r_ic()*, *lambda()*, *d2cheb()*, *dcheb()*, *r_icb()*, *dtemp0()*, *r_cmb()*, *cheb_norm_ic()*, *r()*, *d_costf_init()*, *or3()*, *otemp1()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *logic* (*l_rot_ma()*, *l_anelastic_liquid()*, *l_rot_ic()*, *l_sric()*, *l_cond_ic()*, *l_srma()*): Module containing the logicals that control the run

- *useful* (*random()*): library with several useful subroutines

- *precision_mod*: This module controls the precision used in MagIC

- *physical_parameters* (*phis()*, *kbots()*, *n_imps_max()*, *o_sr()*, *sigma_ratio()*, *epsc()*, *opr()*, *peaks()*, *n_imps()*, *imps()*, *imagcon()*, *ktops()*, *thetas()*, *opm()*, *widths()*, *radratio()*): Module containing the physical parameters

- *cosine_transform* (*costf1()*)

- *lmloop_data* (*llm()*, *ulmmag()*, *llmmag()*, *ulm()*)

- *legendre_grid_to_spec* (*legtf1()*)

- *fft*: This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5

- *constants* (*c_z10_omega_ic()*, *osq4pi()*, *four()*, *three()*, *half()*, *two()*, *y10_norm()*, *one()*, *third()*, *c_z10_omega_ma()*, *pi()*, *zero()*): module containing constants and parameters used in the code.

Variables

- init_fields/**tops** (:,:) *[complex,allocatable/public]*

- init_fields/**l_reset_t** *[logical,public]*
    reset time from startfile ?

- init_fields/**n_start_file** *[integer,public]*
    I/O unit of start_file

- init_fields/**amp_b1** *[real,public]*

- init_fields/**omega_ma2** *[real,public]*

- init_fields/**tshift_ic2** *[real,public]*

- init_fields/**tomega_ma1** *[real,public]*

- init_fields/**tshift_ma2** *[real,public]*

- init_fields/**omegaosz_ic1** *[real,public]*

- init_fields/**inform** *[integer,public]*
    format of start_file

- init_fields/**tshift_ic1** *[real,public]*

- init_fields/**tshift_ma1** *[real,public]*
- init_fields/**init_s1** *[integer,public]*
- init_fields/**amp_s2** *[real,public]*
- init_fields/**omegaosz_ma1** *[real,public]*
- init_fields/**s_bot** (80) *[real,public]*
    input variables for tops,bots
- init_fields/**nrotic** *[integer,public]*
- init_fields/**bpeakbot** *[real,public]*
- init_fields/**tomega_ic1** *[real,public]*
- init_fields/**omega_ic2** *[real,public]*
- init_fields/**tomega_ma2** *[real,public]*
- init_fields/**l_start_file** *[logical,public]*
    taking fields from startfile ?
- init_fields/**tipdipole** *[real,public]*
    adding to symetric field
- init_fields/**amp_s1** *[real,public]*
- init_fields/**scale_v** *[real,public]*
- init_fields/**start_file** *[character,public]*
    name of start_file
- init_fields/**init_b1** *[integer,public]*
- init_fields/**s_top** (80) *[real,public]*
- init_fields/**init_s2** *[integer,public]*
- init_fields/**bots** (:,:) *[complex,allocatable/public]*
- init_fields/**scale_s** *[real,public]*
- init_fields/**scale_b** *[real,public]*
- init_fields/**omega_ic1** *[real,public]*
- init_fields/**n_s_bounds** *[integer,parameter=20/public]*
- init_fields/**omegaosz_ma2** *[real,public]*
- init_fields/**bpeaktop** *[real,public]*
- init_fields/**amp_v1** *[real,public]*
- init_fields/**omega_ma1** *[real,public]*
- init_fields/**init_v1** *[integer,public]*
- init_fields/**nrotma** *[integer,public]*
- init_fields/**omegaosz_ic2** *[real,public]*
- init_fields/**tomega_ic2** *[real,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 *getstartfields()*, *inits()*

Call to *sgefa()*, *sgesl()*

10.5 Pre-calculations

10.5.1 `preCalc.f90`

Quick access

Routines *get_hit_times()*, *precalctimes()*, *precalc()*, *writeinfo()*

Needed modules

- *bext* (*fac_loop()*, *l_curr()*): Module containing the external field parameters

- *num_param*: Module containing numerical and control parameters

- *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

- *truncation* (*n_phi_max()*, *n_r_max()*, *l_max()*, *n_cheb_max()*, *lm_max()*, *nalias()*, *m_max()*, *minc()*, *n_r_ic_max()*, *n_cheb_ic_max()*, *n_theta_max()*): This module defines the grid points and the truncation

- *radial_functions* (*temp0()*, *dddrx()*, *drx()*, *r_surface()*, *kappa()*, *visc()*, *beta()*, *dbeta()*, *radial()*, *dlvisc()*, *rho0()*, *dllambda()*, *sigma()*, *lambda()*, *divktemp0()*, *r_cmb()*, *dlkappa()*, *ddrx()*, *dentropy0()*, *i_costf_init()*, *transportproperties()*, *r()*, *d_costf_init()*, *r_icb()*, *rgrav()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *logic* (*l_anel()*, *l_lcr()*, *l_to()*, *l_mag()*, *l_dt_cmb_field()*, *l_cmb_field()*, *l_anelastic_liquid()*, *l_cond_ic()*, *l_non_rot()*, *l_storebpot()*, *l_storetpot()*, *l_storepot()*, *l_newmap()*, *l_mag_lf()*, *l_save_out()*, *l_time_hits()*, *l_r_field()*, *l_tomovie()*, *l_movie()*, *l_storevpot()*, *l_heat()*): Module containing the logicals that control the run

- *init_fields* (*s_top()*, *s_bot()*, *bots()*, *tops()*, *n_s_bounds()*, *l_reset_t()*)

- *useful* (*logwrite()*): library with several useful subroutines

- *output_data*: This module contains the parameters for output control

- *precision_mod*: This module controls the precision used in MagIC

- *integration* (*rint_r()*): Radial integration functions

- *physical_parameters* (*nvareps()*, *r_lcr()*, *pr()*, *o_sr()*, *radratio()*, *rho_ratio_ma()*, *nvardiff()*, *epsc()*, *buofac()*, *opr()*, *sigma_ratio()*, *nvarvisc()*, *nvarcond()*, *ek()*, *rascaled()*, *opm()*, *prmag()*, *kbots()*, *epsc0()*, *corfac()*, *ra()*, *tmagcon()*, *ekscaled()*, *lffac()*, *mode()*, *interior_model()*, *rho_ratio_ic()*, *ktops()*, *n_r_lcr()*, *polind()*): Module containing the physical parameters

- *constants*: module containing constants and parameters used in the code.

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()*

Chapter 10. Description of the Fortran modules

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** *drx*, *dlvisc*, *rc*, *cheb_int_ic*, *cheb_int*, *temp0*, *divktemp0*, *sigma*, *dtemp0*, *dcheb*, *dllambda*, *dlkappa*, *r*, *beta*, *rgrav*, *d2cheb_ic*, *otemp1*, *or4*, *orho1*, *d_costf_initc*, *epscprof*, *d3cheb*, *d2temp0*, *d_costf1_ic_init*, *dbeta*, *or3*, *visc*, *dentropy0*, *or2*, *ddrx*, *kappa*, *agrav*, *d_costf_init*, *jvarcon*, *dcheb_ic*, *cheb_ic*, *cheb*, *o_r_ic2*, *dr_facc*, *d2cheb*, *lambda*, *o_r_ic*, *orho2*, *d_costf2_ic_init*, *rho0*, *or1*, *dddrx*, *r_ic*, *i_costf_init*, *i_costf_initc*, *i_costf2_ic_init*, *i_costf1_ic_init*, *dr_fac_ic*, *topcond*, *r_cmb*, *dr_fac*, *alpha2*, *alpha1*, *cheb_norm*, *alph1*, *cheb_norm_ic*, *alph2*, *r_icb*, *botcond*, *r_surface*, *ndi_costf2_ic*, *ndd_costf1_ic*, *ndd_costf1*, *ncut*, *ndi_costf1_ic*, *ndi_costf1*, *n_cheb_maxc*, *ndd_costf2_ic*, *n_r_maxc*
>
> **Routines** *getbackground()*, *initialize_radial_functions()*, *transportproperties()*, *radial()*

Needed modules

- *radial_der* (*get_dr()*): Radial derivatives functions

- *algebra* (*sgefa()*, *sgesl()*)

- *num_param* (*alpha()*): Module containing numerical and control parameters

- *cosine_transform* (*costf1()*)

- *matrices* (*s0pivot()*, *s0mat()*): This module contains matricies for internal time step

- *physical_parameters*: Module containing the physical parameters

- *init_costf*

- *truncation* (*n_r_max()*, *n_cheb_max()*, *n_r_ic_max()*): This module defines the grid points and the truncation

- *chebyshev_polynoms_mod*

- *logic* (*l_anel()*, *l_isothermal()*, *l_anelastic_liquid()*, *l_mag()*, *l_heat()*, *l_cond_ic()*, *l_newmap()*): Module containing the logicals that control the run

- *constants* (*two()*, *one()*, *four()*, *three()*, *sq4pi()*, *half()*): module containing constants and parameters used in the code.

Variables

- radial_functions/**drx** (:) *[real,allocatable/public]*

- radial_functions/**i_costf_initc** (:) *[integer,allocatable/public]*

- radial_functions/**o_r_ic2** (:) *[real,allocatable/public]*

- radial_functions/**n_r_maxc** *[integer,public]*
  Number of radial points

- radial_functions/**d_costf1_ic_init** (:) *[real,allocatable/public]*

- radial_functions/**beta** (:) *[real,allocatable/public]*

- radial_functions/**dbeta** (:) *[real,allocatable/public]*

- radial_functions/**dlvisc** (:) *[real,allocatable/public]*

- radial_functions/**i_costf1_ic_init** (:) *[integer,allocatable/public]*

- radial_functions/**rho0** (:) *[real,allocatable/public]*

- radial_functions/**sigma** (:) *[real,allocatable/public]*

- radial_functions/**d2cheb** (:,:) *[real,allocatable/public]*

- radial_functions/**i_costf2_ic_init** (:) *[integer,allocatable/public]*

- radial_functions/**alpha1** *[real,public]*
  Input parameter for non-linear map to define degree of spacing (0.0:2.0)

- radial_functions/**dcheb** (:,:) *[real,allocatable/public]*

- radial_functions/**cheb_norm_ic** *[real,public]*
  Chebyshev normalisation for IC

- radial_functions/**or3** (:) *[real,allocatable/public]*

- radial_functions/**cheb_int_ic** (:) *[real,allocatable/public]*

- radial_functions/**temp0** (:) *[real,allocatable/public]*

- radial_functions/**d2cheb_ic** (:,:) *[real,allocatable/public]*

- radial_functions/**jvarcon** (:) *[real,allocatable/public]*

- radial_functions/**visc** (:) *[real,allocatable/public]*

- radial_functions/**lambda** (:) *[real,allocatable/public]*

- radial_functions/**dr_fac** *[real,public]*
    $2/d$, where $d = r_o - r_i$

- radial_functions/**n_cheb_maxc** *[integer,public]*
    Number of Chebyshevs

- radial_functions/**dentropy0** (:) *[real,allocatable/public]*

- radial_functions/**ndd_costf1_ic** *[integer,public]*
    Radii for transform

- radial_functions/**epscprof** (:) *[real,allocatable/public]*

- radial_functions/**divktemp0** (:) *[real,allocatable/public]*

- radial_functions/**ndd_costf1** *[integer,public]*
    Radii for transform

- radial_functions/**cheb_ic** (:,:) *[real,allocatable/public]*

- radial_functions/**dcheb_ic** (:,:) *[real,allocatable/public]*

- radial_functions/**cheb** (:,:) *[real,allocatable/public]*

- radial_functions/**alph1** *[real,public]*
    Input parameter for non-linear map to define degree of spacing (0.0:2.0)

- radial_functions/**or1** (:) *[real,allocatable/public]*

- radial_functions/**d2temp0** (:) *[real,allocatable/public]*

- radial_functions/**or4** (:) *[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/**rgrav** (:) *[real,allocatable/public]*

- radial_functions/**dr_facc** (:) *[real,allocatable/public]*

- radial_functions/**orho1** (:) *[real,allocatable/public]*

- radial_functions/**cheb_norm** *[real,public]*
    Chebyshev normalisation

- radial_functions/**or2** (:) *[real,allocatable/public]*

- radial_functions/**dllambda** (:) *[real,allocatable/public]*

- radial_functions/**ndi_costf1** *[integer,public]*
    Radii for transform

- radial_functions/**botcond** *[real,public]*
    Heat flux at IC boundary

- radial_functions/**orho2** (:) *[real,allocatable/public]*

- radial_functions/**topcond** *[real,public]*
    Heat flux at OC boundary

- radial_functions/**r_cmb** *[real,public]*
    OC radius

- radial_functions/**o_r_ic** (:) *[real,allocatable/public]*

- radial_functions/**dr_fac_ic** *[real,public]*
    For IC: $2/(2r_i)$

- radial_functions/**ddrx** (:) *[real,allocatable/public]*

- radial_functions/**ndi_costf2_ic** *[integer,public]*
    Radii for transform

- radial_functions/**i_costf_init** (:) *[integer,allocatable/public]*

- radial_functions/**d_costf_initc** (:) *[real,allocatable/public]*

- radial_functions/**r_icb** *[real,public]*
    IC radius

- radial_functions/**alpha2** *[real,public]*
    Input parameter for non-linear map to define central point of different spacing (-1.0:1.0)

- radial_functions/**dddrx** (:) *[real,allocatable/public]*

- radial_functions/**rc** (:) *[real,allocatable/public]*

- radial_functions/**d3cheb** (:,:) *[real,allocatable/public]*

- radial_functions/**d_costf_init** (:) *[real,allocatable/public]*

- radial_functions/**kappa** (:) *[real,allocatable/public]*

- radial_functions/**ndd_costf2_ic** *[integer,public]*
    Radii for transform

- radial_functions/**cheb_int** (:) *[real,allocatable/public]*

- radial_functions/**agrav** (:) *[real,allocatable/public]*

- radial_functions/**ncut** *[integer,public]*
    Truncation

- radial_functions/**dtemp0** (:) *[real,allocatable/public]*

- radial_functions/**d_costf2_ic_init** (:) *[real,allocatable/public]*

- radial_functions/**dlkappa** (:) *[real,allocatable/public]*

- radial_functions/**otemp1** (:) *[real,allocatable/public]*

- radial_functions/**r** (:) *[real,allocatable/public]*

- radial_functions/**r_surface** *[real,public]*
    Surface radius for extrapolation

- radial_functions/**r_ic** (:) *[real,allocatable/public]*

- radial_functions/**ndi_costf1_ic** *[integer,public]*
    Radii for transform

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()*, *init_costf1()*, *getbackground()*, *get_chebs_even()*, *init_costf2()*

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** *d_mc2m*, *d_m*, *dtheta1s*, *dtheta4s*, *hdif_b*, *dphi0*, *plm*, *osn2*, *dtheta3a*, *dpl0eq*, *sintheta*, *d_lp1*, *cosn2*, *dplm*, *hdif_s*, *theta*, *o_sin_theta*, *o_sin_theta_e2*, *dtheta2a*, *d_l*, *dtheta2s*, *osn1*, *dtheta1a*, *gauss*, *costheta*, *dtheta4a*, *dlh*, *phi*, *dtheta3s*, *theta_ord*, *dphi*, *sn2*, *dphi02*, *wplm*, *hdif_v*, *lstopp*, *lstart*, *lmoddp*, *lstartp*, *n_theta_cal2ord*, *lmodd*, *lstop*
>
> **Routines** *initialize_horizontal_data()*, *horizontal()*, *gauleg()*

Needed modules

> • *blocking* (*lm2l()*, *lmp2lm()*, *lmp2l()*, *lm2m()*): Module containing blocking information
>
> • *plms_theta* (*plm_theta()*, *plm_thetaas()*)

- *num_param* (*difnu()*, *difkap()*, *ldif()*, *ldifexp()*, *difeta()*): Module containing numerical and control parameters

- *precision_mod*: This module controls the precision used in MagIC

- *physical_parameters* (*ek()*): Module containing the physical parameters

- *truncation* (*n_theta_max()*, *n_phi_max()*, *l_max()*, *n_m_max()*, *lm_max()*, *m_max()*, *minc()*, *lmp_max()*): This module defines the grid points and the truncation

- *radial_functions* (*r_cmb()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *logic* (*l_non_rot()*): Module containing the logicals that control the run

- *fft*: This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5

- *constants* (*two()*, *half()*, *pi()*, *one()*, *zero()*): module containing constants and parameters used in the code.

Variables

- horizontal_data/**dpl0eq** (:) *[real,allocatable/public]*

- horizontal_data/**osn1** (:) *[real,allocatable/public]*

- horizontal_data/**phi** (:) *[real,allocatable/public]*

- horizontal_data/**lstop** (:) *[integer,allocatable/public]*

- horizontal_data/**dplm** (:,:) *[real,allocatable/public]*

- horizontal_data/**costheta** (:) *[real,allocatable/public]*

- horizontal_data/**o_sin_theta** (:) *[real,allocatable/public]*

- horizontal_data/**d_m** (:) *[real,allocatable/public]*

- horizontal_data/**lstartp** (:) *[integer,allocatable/public]*

- horizontal_data/**hdif_s** (:) *[real,allocatable/public]*

- horizontal_data/**o_sin_theta_e2** (:) *[real,allocatable/public]*

- horizontal_data/**dtheta3a** (:) *[real,allocatable/public]*

- horizontal_data/**dtheta3s** (:) *[real,allocatable/public]*

- horizontal_data/**theta** (:) *[real,allocatable/public]*

- horizontal_data/**dtheta2a** (:) *[real,allocatable/public]*

- horizontal_data/**n_theta_cal2ord** (:) *[integer,allocatable/public]*

- horizontal_data/**hdif_b** (:) *[real,allocatable/public]*

- horizontal_data/**theta_ord** (:) *[real,allocatable/public]*

- horizontal_data/**dphi0** (:) *[complex,allocatable/public]*

- horizontal_data/**d_l** (:) *[real,allocatable/public]*

- horizontal_data/**lstopp** (:) *[integer,allocatable/public]*

- horizontal_data/**lstart** (:) *[integer,allocatable/public]*

- horizontal_data/**dtheta1a** (:) *[real,allocatable/public]*

- `horizontal_data/`**dtheta4a** (:) *[real,allocatable/public]*

- `horizontal_data/`**d_lp1** (:) *[real,allocatable/public]*

- `horizontal_data/`**dlh** (:) *[real,allocatable/public]*

- `horizontal_data/`**sintheta** (:) *[real,allocatable/public]*

- `horizontal_data/`**lmodd** (:) *[logical,allocatable/public]*

- `horizontal_data/`**wplm** (:,:) *[real,allocatable/public]*

- `horizontal_data/`**osn2** (:) *[real,allocatable/public]*

- `horizontal_data/`**dphi** (:) *[complex,allocatable/public]*

- `horizontal_data/`**hdif_v** (:) *[real,allocatable/public]*

- `horizontal_data/`**gauss** (:) *[real,allocatable/public]*

- `horizontal_data/`**dtheta2s** (:) *[real,allocatable/public]*

- `horizontal_data/`**sn2** (:) *[real,allocatable/public]*

- `horizontal_data/`**d_mc2m** (:) *[real,allocatable/public]*

- `horizontal_data/`**dtheta1s** (:) *[real,allocatable/public]*

- `horizontal_data/`**plm** (:,:) *[real,allocatable/public]*

- `horizontal_data/`**dphi02** (:) *[complex,allocatable/public]*

- `horizontal_data/`**dtheta4s** (:) *[real,allocatable/public]*

- `horizontal_data/`**cosn2** (:) *[real,allocatable/public]*

- `horizontal_data/`**lmoddp** (:) *[logical,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_rloc*, *dsdt_rloc*, *dvsrlm_lmloc*, *dvxbhlm_lmloc*, *dvxbhlm_rloc*, *dwdt_lmloc*, *dzdt_rloc*, *dbdt_lmloc*, *dzdt_lmloc*, *dpdt_rloc*, *djdt_lmloc*, *djdt_rloc*, *dsdt_lmloc*, *dvsrlm_rloc*, *dpdt_lmloc*, *dwdt_rloc*

> **Routines** *step_time()*, *initialize_step_time()*, *check_time_hits()*

Needed modules

- *blocking* (*nlmbs()*, *lmstartb()*, *lmstopb()*): Module containing blocking information

- *communications* (*lo2r_redist_wait()*, *lo2r_b()*, *lo2r_aj()*, *lo2r_s()*, *r2lo_redist()*, *get_global_sum()*, *lo2r_p()*, *lo2r_redist_start()*, *lo2r_w()*, *lm2r_type()*, *lo2r_z()*)

- *lmloop_mod* (*lmloop()*)

- *fields*: This module contains the potential fields and their radial derivatives

- *truncation* (*n_r_max()*, *l_max()*, *lm_maxmag()*, *n_r_maxmag()*, *lm_max()*, *l_maxmag()*, *lmp_max()*): This module defines the grid points and the truncation

- *num_param* (*runtimelimit()*, *dtmax()*, *tscale()*, *dtmin()*, *runtime()*, *tend()*, *n_time_steps()*, *alpha()*): Module containing numerical and control parameters

- *timing*: Useful functions for time-stepping

- *radialloop* (*radialloopg()*)

- *charmanip* (*capitalize()*, *dble2str()*): This module contains several useful routines to manipule character strings

- *output_mod* (*output()*)

- *radial_data* (*nrstartmag()*, *nrstopmag()*, *n_r_icb()*, *nrstart()*, *n_r_cmb()*, *nrstop()*)

- *output_data* (*log_file()*, *n_r_fields()*, *n_t_rst()*, *n_t_movie()*, *n_graph_step()*, *t_r_field()*, *n_tpot_step()*, *t_spec()*, *n_log_step()*, *n_t_tpot()*, *n_t_to()*, *n_time_hits()*, *tag()*, *n_tos()*, *t_movie()*, *n_t_vpot()*, *n_toz_step()*, *n_cmb_step()*, *t_tomovie()*, *n_vpot_step()*, *graph_mpi_fh()*, *n_log_file()*, *n_bpot_step()*, *n_t_tomovie()*, *n_t_cmb()*, *n_logs()*, *n_spec_step()*, *n_cmbs()*, *n_vpots()*, *l_graph_time()*, *n_bpots()*, *n_tomovie_step()*, *n_t_bpot()*, *t_to()*, *nlf()*, *n_t_log()*, *t_rst()*, *n_tomovie_frames()*, *t_graph()*, *n_movie_frames()*, *t_log()*, *n_t_r_field()*, *n_specs()*, *n_t_toz()*, *t_toz()*, *n_stores()*, *t_cmb()*, *n_tpots()*, *graph_file()*, *n_t_spec()*, *t_vpot()*, *n_r_field_step()*, *n_rsts()*, *t_bpot()*, *n_movie_step()*, *n_to_step()*, *n_graphs()*, *n_tozs()*, *n_rst_step()*, *t_tpot()*, *n_t_graph()*): This module contains the parameters for output control

- *logic* (*l_b_nl_cmb()*, *l_perppar()*, *l_to()*, *l_storebpot()*, *l_storetpot()*, *l_fluxprofs()*, *l_true_time()*, *l_time_hits()*, *lverbose()*, *l_tomovie()*, *l_mag_lf()*, *l_storevpot()*, *l_heat()*, *l_ht()*, *l_dtrmagspec()*, *l_hel()*, *l_viscbccalc()*, *l_mag()*, *l_conv()*, *l_dtbmovie()*, *l_cmb_field()*, *l_save_out()*, *l_r_field()*, *l_htmovie()*, *l_dtb()*, *l_rms()*, *l_movie()*, *l_b_nl_icb()*, *l_runtimelimit()*): Module containing the logicals that control the run

- *fieldslast*: This module contains time-derivaties array of the previous time-step They are needed in the time-stepping scheme.

- *useful* (*l_correct_step()*, *logwrite()*, *safeopen()*, *safeclose()*): library with several useful subroutines

- *courant_mod* (*dt_courant()*)

- *precision_mod*: This module controls the precision used in MagIC

- *parallel_mod*: This module contains the blocking information

- *movie_data* (*t_movies()*)

- *lmloop_data* (*llm()*, *ulmmag()*, *llmmag()*, *lm_per_rank()*, *ulm()*, *lm_on_last_rank()*)

- *nonlinear_bcs* (*get_b_nl_bcs()*)

- *constants* (*half()*, *one()*, *zero()*): module containing constants and parameters used in the code.

Variables

- step_time_mod/**dwdt_rloc** (:,:) *[complex,private/allocatable]*

- step_time_mod/**dbdt_lmloc** (:,:) *[complex,private/allocatable]*

- step_time_mod/**dvsrlm_lmloc** (:,:) *[complex,private/allocatable]*

- step_time_mod/**dpdt_lmloc** (:,:) *[complex,private/allocatable]*

- step_time_mod/**djdt_rloc** (:,:) *[complex,private/allocatable]*

- step_time_mod/**dvsrlm_rloc** (:,:) *[complex,private/allocatable]*

- step_time_mod/**dbdt_rloc** (:,:) *[complex,private/target/allocatable]*

- step_time_mod/**dsdt_rloc** (:,:) *[complex,private/allocatable]*

- step_time_mod/**dpdt_rloc** (:,:) *[complex,private/allocatable]*

- step_time_mod/**dvxbhlm_lmloc** (:,:) *[complex,private/allocatable]*

- step_time_mod/**dsdt_lmloc** (:,:) *[complex,private/allocatable]*

- step_time_mod/**dzdt_lmloc** (:,:) *[complex,private/allocatable]*

- step_time_mod/**djdt_lmloc** (:,:) *[complex,private/allocatable]*

- step_time_mod/**dvxbhlm_rloc** (:,:) *[complex,private/allocatable]*

- step_time_mod/**dwdt_lmloc** (:,:) *[complex,private/allocatable]*

- step_time_mod/**dzdt_rloc** (:,:) *[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

- *blocking* (*nfs()*): Module containing blocking information
- *num_param* (*delxr2()*, *delxh2()*, *courfac()*, *alffac()*): Module containing numerical and control parameters
- *horizontal_data* (*osn2()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

---

- *parallel_mod*: This module contains the blocking information

- *radial_data* (*nrstart()*, *nrstop()*)

- *truncation* (*n_phi_max()*, *nrp()*): This module defines the grid points and the truncation

- *radial_functions* (*or4()*, *orho2()*, *or2()*, *orho1()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *logic* (*l_mag_lf()*, *l_mag()*, *l_mag_kin()*): Module containing the logicals that control the run

- *useful* (*logwrite()*): library with several useful subroutines

- *precision_mod*: This module controls the precision used in MagIC

- *physical_parameters* (*opm()*, *lffac()*): Module containing the physical parameters

- *constants* (*two()*, *one()*, *half()*): module containing constants and parameters used in the code.

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_seq(), *do_iteration_thetablocking_openmp()*

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** *lnegtime()*, *ltimelimit()*, *time2ms()*, *writetime()*, *subtime()*, *addtime()*, *ms2time()*, *walltime()*, *meantime()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- `mpi`
- *parallel_mod* (rank()): This module contains the blocking information

Variables

- `timing`/**msechour** *[integer,private/parameter=3600000]*
- `timing`/**msecminute** *[integer,private/parameter=60000]*
- `timing`/**msecsecond** *[integer,private/parameter=1000]*

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 `subtime()`, `addtime()`, `step_time()`, `walltime()`, `meantime()`

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 `subtime()`, `lnegtime()`, `addtime()`, `step_time()`, `meantime()`

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

- *blocking* (*lmstartb()*, *lmstopb()*): Module containing blocking information

- *communications* (*lo2r_aj()*, *lo2r_z()*, *lo2r_redist_start()*, *lo2r_p()*, *get_global_sum()*, *lo2r_w()*, *lo2r_s()*, *lo2r_b()*)

- *updatez_mod* (*initialize_updatez()*, *updatez()*)

- *radial_data* (*n_r_cmb()*, *n_r_icb()*)

- *matrices* (*lzmat()*, *lbmat()*, *lwpmat()*, *lz10mat()*, *lsmat()*): This module contains matricies for internal time step

- *output_data* (*log_file()*, *nlf()*): This module contains the parameters for output control

- *logic* (*l_heat()*, *l_anelastic_liquid()*, *l_mag()*, *lverbose()*, *l_conv()*): Module containing the logicals that control the run

- *fieldslast*: This module contains time-derivaties array of the previous time-step They are needed in the time-stepping scheme.

- *fields*: This module contains the potential fields and their radial derivatives

- omp_lib

- *updatewp_mod* (*initialize_updatewp()*, *updatewp()*)

- *truncation* (*lm_max()*, *l_max()*, *n_r_maxmag()*, *n_r_max()*): This module defines the grid points and the truncation

- *precision_mod*: This module controls the precision used in MagIC

- *updates_mod* (*updates()*, *updates_ala()*, *initialize_updates()*)

- *timing* (*subtime()*, *walltime()*, *writetime()*): Useful functions for time-stepping

- *parallel_mod* (rank()): This module contains the blocking information

- *updateb_mod* (*initialize_updateb()*, *updateb()*)

- *lmloop_data* (*llm()*, *ulmmag()*, *llmmag()*, *ulm()*)

- debugging (debug_write())

- *useful* (*safeopen()*, *safeclose()*): library with several useful subroutines

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*, *n_time_step*)

> 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
> - **n_time_step** *[integer,in]*

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 *pre*, *buo*, *dif*

Routines *finalize_updatewp()*, *updatewp()*, *get_wpmat()*, *initialize_updatewp()*

Needed modules

- *blocking* (*lo_map()*, *lmstartb()*, *st_sub_map()*, *lmstopb()*, *nlmbs()*, *st_map()*, *lo_sub_map()*): Module containing blocking information
- *algebra* (*sgefa()*, *cgeslml()*)

- *num_param* (*alpha()*): Module containing numerical and control parameters

- *horizontal_data* (*dlh()*, *hdif_v()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

- *radial_data* (*n_r_cmb()*, *n_r_icb()*)

- *matrices* (*wppivot()*, *wpmat_fac()*, *lwpmat()*, *wpmat()*): This module contains matricies for internal time step

- *communications* (*get_global_sum()*)

- *truncation* (*lm_max()*, *n_cheb_max()*, *n_r_max()*): This module defines the grid points and the truncation

- *radial_functions* (*dddrx()*, *dlvisc()*, *drx()*, *cheb()*, *d3cheb()*, *or1()*, *dcheb()*, *beta()*, *ddrx()*, *cheb_norm()*, *d2cheb()*, *visc()*, *i_costf_init()*, *rgrav()*, *d_costf_init()*, *dbeta()*, *agrav()*, *or2()*, *rho0()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *logic* (*l_update_v()*, *l_rmstest()*): Module containing the logicals that control the run

- omp_lib

- *radial_der* (*get_dr()*, *get_dddr()*): Radial derivatives functions

- *precision_mod*: This module controls the precision used in MagIC

- *physical_parameters* (*ra()*, *ktopv()*, *kbotv()*): Module containing the physical parameters

- *cosine_transform* (*costf1()*)

- *lmloop_data* (*llm()*, *ulm()*)

- *rms_helpers* (*hint2pol()*): This module contains several useful subroutines required to compute RMS diagnostics

- *parallel_mod* (*chunksize()*): This module contains the blocking information

- *constants* (*two()*, *one()*, *four()*, *three()*, *third()*, *half()*, *zero()*): module containing constants and parameters used in the code.

- *rms* (*dtvpol2hint()*, *dtvpolas2hint()*, *dtvpollmr()*, *difpolas2hint()*, *difpol2hint()*, *difpollmr()*): This module contains the global array used when RMS force balance is requested

Variables

- updatewp_mod/**worka** (:,:) *[complex,private/allocatable]*

- updatewp_mod/**maxthreads** *[integer,private]*

- updatewp_mod/**rhs1** (:,:,:) *[complex,private/allocatable]*

- updatewp_mod/**dif** (:) *[complex,private/allocatable]*

- updatewp_mod/**buo** (:) *[complex,private/allocatable]*

- updatewp_mod/**pre** (:) *[complex,private/allocatable]*

- updatewp_mod/**workb** (:,:) *[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_wpmat()*, *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()`

10.7.3 `updateZ.f90`

Quick access

> **Variables** `workc`, `worka`, `workb`, `rhs1`, `maxthreads`
>
> **Routines** `updatez()`, `initialize_updatez()`, `get_zmat()`, `get_z10mat()`

Needed modules

- `blocking` (`lo_map()`, `lmstartb()`, `st_sub_map()`, `lmstopb()`, `nlmbs()`, `st_map()`, `lo_sub_map()`): Module containing blocking information

- `communications` (`get_global_sum()`)

- `num_param` (`alpha()`, `amstart()`): Module containing numerical and control parameters

- `parallel_mod`: This module contains the blocking information

- `matrices` (`lzmat()`, `z10mat()`, `zmat()`, `lz10mat()`, `zpivot()`, `zmat_fac()`, `z10pivot()`, `z10mat_fac()`): This module contains matricies for internal time step

- `truncation` (`n_r_max()`, `l_max()`, `n_cheb_max()`, `lm_max()`): This module defines the grid points and the truncation

- `init_fields`

- `rms` (`dtvtor2hint()`, `diftoras2hint()`, `dtvtoras2hint()`, `diftor2hint()`): This module contains the global array used when RMS force balance is requested

- `cosine_transform` (`costf1()`)

- `rms_helpers` (`hint2pol()`, `hint2tor()`): This module contains several useful subroutines required to compute RMS diagnostics

- `algebra` (`cgesl()`, `sgefa()`, `cgeslml()`)

- `radial_data` (`n_r_cmb()`, `n_r_icb()`)

- `radial_der` (`get_ddr()`): Radial derivatives functions

- `radial_functions` (`d2cheb()`, `drx()`, `cheb()`, `or1()`, `dcheb()`, `ddrx()`, `cheb_norm()`, `r_cmb()`, `visc()`, `i_costf_init()`, `or2()`, `r()`, `r_icb()`, `dbeta()`, `beta()`, `dlvisc()`, `rho0()`, `d_costf_init()`): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- `logic` (`l_rot_ma()`, `l_to()`, `l_correct_ame()`, `l_rmstest()`, `l_z10mat()`, `l_rot_ic()`, `l_sric()`, `l_update_v()`, `l_correct_amz()`, `l_srma()`): Module containing the logicals that control the run

- `omp_lib`

- `torsional_oscillations` (`ddzasl()`): This module contains information for TO calculation and output

- `outrot` (`get_angular_moment()`)

- `precision_mod`: This module controls the precision used in MagIC

- *horizontal_data* (*dlh()*, *hdif_v()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

- *physical_parameters* (*ktopv()*, *lffac()*, *kbotv()*): Module containing the physical parameters

- *lmloop_data* (*llm()*, *ulm()*)

- *constants* (*c_lorentz_ic()*, *c_z10_omega_ic()*, *c_moi_ma()*, *half()*, *c_moi_oc()*, *c_dt_z10_ma()*, *two()*, *y11_norm()*, *one()*, *c_lorentz_ma()*, *c_dt_z10_ic()*, *four()*, *y10_norm()*, *zero()*, *c_moi_ic()*, *c_z10_omega_ma()*): module containing constants and parameters used in the code.

Variables

- updatez_mod/**workc** (:,:) *[complex,private/allocatable]*

- updatez_mod/**worka** (:,:) *[complex,private/allocatable]*

- updatez_mod/**maxthreads** *[integer,private]*

- updatez_mod/**rhs1** (:,:,:) *[complex,private/allocatable]*

- updatez_mod/**workb** (:,:) *[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()*, *cgeslml()*, *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()*, *updateS()*, *get_s0mat()*, *get_sMat()*, *updateS_ala()*

Needed modules

- *blocking* (*lo_map()*, *lmstartb()*, *lmstopb()*, *nlmbs()*, *lo_sub_map()*, *st_map()*): Module containing blocking information

- *algebra* (*sgefa()*, *sgesl()*, *cgeslml()*)

- *num_param* (*alpha()*): Module containing numerical and control parameters

- *horizontal_data* (*dlh()*, *hdif_s()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

- *matrices* (*s0pivot()*, *spivot()*, *s0mat_fac()*, *smat_fac()*, *smat()*, *lsmat()*, *s0mat()*): This module contains matricies for internal time step

- *radial_data* (*n_r_cmb()*, *n_r_icb()*)

- *truncation* (*n_r_max()*, *n_cheb_max()*, *lm_max()*): This module defines the grid points and the truncation

- *radial_functions* (*temp0()*, *d2cheb()*, *drx()*, *cheb()*, *dcheb()*, *or1()*, *kappa()*, *dlkappa()*, *ddrx()*, *dtemp0()*, *dentropy0()*, *i_costf_init()*, *beta()*, *otemp1()*, *orho1()*, *cheb_norm()*, *or2()*, *d_costf_init()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *logic* (*l_update_s()*, *l_anelastic_liquid()*): Module containing the logicals that control the run

- omp_lib

- *init_fields* (*bots()*, *tops()*)

- *radial_der* (*get_drns()*, *get_ddr()*): Radial derivatives functions

- *precision_mod*: This module controls the precision used in MagIC

- *physical_parameters* (*ktops()*, *kbots()*, *opr()*): Module containing the physical parameters

- *cosine_transform* (*costf1()*)

- *lmloop_data* (*llm()*, *ulm()*)

- *parallel_mod* (rank(), *chunksize()*): This module contains the blocking information

- *constants* (*two()*, *half()*, *one()*, *zero()*): module containing constants and parameters used in the code.

Variables

- updates_mod/**worka** (:,:) *[complex,private/allocatable]*

- updates_mod/**maxthreads** *[integer,private]*

- updates_mod/**rhs1** (:,:,:) *[complex,private/allocatable]*

- updates_mod/**workb** (:,:) *[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 *rhs2*

Routines *initialize_updateb()*, *updateb()*, *get_bmat()*

Needed modules

- *blocking* (*lo_map()*, *lmstartb()*, *st_sub_map()*, *lmstopb()*, *nlmbs()*, *lo_sub_map()*, *st_map()*): Module containing blocking information
- *num_param* (*alpha()*): Module containing numerical and control parameters
- *parallel_mod* (rank(), *chunksize()*): This module contains the blocking information
- *matrices* (*jpivot()*, *bmat()*, *bmat_fac()*, *bpivot()*, *lbmat()*, *jmat()*, *jmat_fac()*): This module contains matricies for internal time step
- *truncation* (*n_r_ic_maxmag()*, *n_r_max()*, *n_cheb_max()*, *lm_max()*, *n_r_totmag()*, *n_r_maxmag()*, *n_r_tot()*, *n_cheb_ic_max()*, *n_r_ic_max()*): This module defines the grid points and the truncation
- *rms* (*dtbpollmr()*, *dtbpol2hint()*, *dtbtoras2hint()*, *dtbtor2hint()*, *dtbpolas2hint()*): This module contains the global array used when RMS force balance is requested
- *init_fields* (*bpeaktop()*, *bpeakbot()*)
- *cosine_transform* (*costf1()*)
- *rms_helpers* (*hint2pol()*, *hint2tor()*): This module contains several useful subroutines required to compute RMS diagnostics

- *bext*: Module containing the external field parameters

- *algebra* (*sgefa()*, *cgeslml()*)

- *radial_data* (*n_r_cmb()*, *n_r_icb()*)

- *radial_der* (*get_drns()*, *get_ddr()*): Radial derivatives functions

- *radial_functions* (*cheb_norm_ic()*, *cheb_ic()*, *dcheb_ic()*, *drx()*, *cheb()*, *d2cheb_ic()*, *or1()*, *d_costf1_ic_init()*, *or2()*, *cheb_norm()*, *lambda()*, *i_costf1_ic_init()*, *dllambda()*, *dr_fac_ic()*, *d2cheb()*, *d_costf2_ic_init()*, *i_costf2_ic_init()*, *o_r_ic()*, *dcheb()*, *ddrx()*, *r_cmb()*, *i_costf_init()*, *r()*, *d_costf_init()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *logic* (*l_lcr()*, *l_update_b()*, *l_b_nl_cmb()*, *l_rot_ic()*, *l_b_nl_icb()*, *l_mag_nl()*, *l_cond_ic()*): Module containing the logicals that control the run

- omp_lib

- *radial_der_even* (*get_ddr_even()*)

- *precision_mod*: This module controls the precision used in MagIC

- *horizontal_data* (*d_l()*, *dlh()*, *dphi()*, *hdif_b()*, *d_lp1()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

- *physical_parameters* (*sigma_ratio()*, *conductance_ma()*, *imagcon()*, *ktopb()*, *tmagcon()*, *o_sr()*, *opm()*, *kbotb()*, *n_r_lcr()*): Module containing the physical parameters

- *lmloop_data* (*ulmmag()*, *llmmag()*)

- *constants* (*two()*, *one()*, *three()*, *half()*, *pi()*, *zero()*): module containing constants and parameters used in the code.

Variables

- updateb_mod/**rhs2** (:,:,:) *[complex,private/allocatable]*

- updateb_mod/**worka** (:,:) *[complex,private/allocatable]*

- updateb_mod/**maxthreads** *[integer,private]*

- updateb_mod/**rhs1** (:,:,:) *[complex,private/allocatable]*

- updateb_mod/**workb** (:,:) *[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()*, *hint2pol()*, *hint2tor()*

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 *initialize_radialloop()*, *radialloopg()*, *finalize_radialloop()*

Needed modules

- *blocking* (*sizethetab()*, *nthetabs()*): Module containing blocking information
- riterthetablocking_seq_mod (riterthetablocking_seq_t())
- *physical_parameters* (*ktopv()*, *kbotv()*): Module containing the physical parameters
- *radial_data* (*nrstartmag()*, *nrstopmag()*, *n_r_icb()*, *nrstart()*, *n_r_cmb()*, *nrstop()*)
- *truncation* (*lm_max()*, *l_max()*, *lmp_max()*, *lm_maxmag()*, *l_maxmag()*): This module defines the grid points and the truncation
- *graphout_mod* (*graphout_mpi_header()*)
- *logic* (*l_rot_ma()*, *l_mag_lf()*, *l_mag()*, *l_mag_kin()*, *lverbose()*, *l_rot_ic()*, *l_dtb()*, *l_mag_nl()*, *l_cond_ic()*, *l_cond_ma()*): Module containing the logicals that control the run
- *riterthetablocking_openmp_mod* (riterthetablocking_openmp_t())
- *precision_mod*: This module controls the precision used in MagIC
- *riteration_mod* (riteration_t())
- *riterthetablocking_mod* (riterthetablocking_t())
- *parallel_mod* (*n_procs()*, rank()): This module contains the blocking information

- *fft*: This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5

- *constants* (*zero()*): module containing constants and parameters used in the code.

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*, *lperpparcalc*, *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*, *fvisclmr*, *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

    - % **lperpparcalc** *[logical]*

    - % **nbc** *[integer]*

    - % **lderiv** *[logical]*

    - % **lhelcalc** *[logical]*

    - % **lviscbccalc** *[logical]*

    - % **lmagnlbc** *[logical]*

    - % **dtrkc** *[real]*

    - % **nr** *[integer]*

    - % **ltocalc** *[logical]*

    - % **l_frame** *[logical]*

    - % **lfluxprofcalc** *[logical]*

    - % **ltonext2** *[logical]*

    - % **l_cour** *[logical]*

    - % **dthkc** *[real]*

    - % **l_graph** *[logical]*

    - % **ltonext** *[logical]*

    - % **isradialboundarypoint** *[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*)

    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]*

10.8.3 `rIterThetaBlocking.f90`

Quick access

Routines `set_thetablocking()`, `allocate_common_arrays()`, `transform_to_lm_space()`, `deallocate_common_arrays()`, `transform_to_grid_space()`

Needed modules

- `blocking` (`nfs()`): Module containing blocking information

- `radial_data` (`nrstart()`, `n_r_cmb()`, `n_r_icb()`, `nrstop()`)

- `truncation` (`lm_max()`, `l_max()`, `n_theta_maxstr()`, `lm_maxmag()`, `n_phi_maxstr()`, `nrp()`, `lmp_max_dtb()`, `lmp_max()`, `n_r_maxstr()`): This module defines the grid points and the truncation

- `radial_functions` (`or2()`, `orho1()`): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- `leg_helper_mod` (leg_helper_t())

- `logic` (`l_anel()`, `l_rot_ma()`, `l_b_nl_cmb()`, `l_mag()`, `l_mag_kin()`, `l_conv()`, `l_conv_nl()`, `l_ht()`, `l_mag_nl()`, `l_cond_ic()`, `l_to()`, `l_cond_ma()`, `l_mag_lf()`, `l_dtb()`, `l_movie_oc()`, `l_rot_ic()`, `l_store_frame()`, `l_heat()`, `l_b_nl_icb()`): Module containing the logicals that control the run

- `legendre_spec_to_grid` (`legtfg()`, `legtfgnomag()`)

- `precision_mod`: This module controls the precision used in MagIC

- `riteration_mod` (riteration_t())

- `physical_parameters` (`n_r_lcr()`, `kbots()`, `ktops()`): Module containing the physical parameters

- `nonlinear_bcs` (`v_rigid_boundary()`)

- `legendre_grid_to_spec`

- `fft`: This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5

- `nonlinear_lm_mod` (nonlinear_lm_t())

- grid_space_arrays_mod(grid_space_arrays_t())

Types

- **type** riterthetablocking_mod/**unknown_type**

    Type fields

    - % **btvpsn2lm** (*) *[complex,allocatable]*
    - % **btvzlm** (*) *[complex,allocatable]*
    - % **brvzlm** (*) *[complex,allocatable]*
    - % **brvtlm** (*) *[complex,allocatable]*
    - % **btvzsn2lm** (*) *[complex,allocatable]*
    - % **brvplm** (*) *[complex,allocatable]*
    - % **bpvtlm** (*) *[complex,allocatable]*
    - % **btvpcotlm** (*) *[complex,allocatable]*
    - % **btvzcotlm** (*) *[complex,allocatable]*
    - % **bpvtsn2lm** (*) *[complex,allocatable]*
    - % **btvplm** (*) *[complex,allocatable]*
    - % **bpvtcotlm** (*) *[complex,allocatable]*
    - % **btvrlm** (*) *[complex,allocatable]*
    - % **bpvrlm** (*) *[complex,allocatable]*

- **type** riterthetablocking_mod/**unknown_type**

    Type fields

    - % **dzrstrlm** (*) *[real,allocatable]*
    - % **dzcorlm** (*) *[real,allocatable]*
    - % **dzastrlm** (*) *[real,allocatable]*
    - % **dzlflm** (*) *[real,allocatable]*

- **type** riterthetablocking_mod/**unknown_type**

    Type fields

    - % **leg_helper** *[leg_helper_t]*
    - % **dtb_arrays** *[dtb_arrays_t]*
    - % **bzlast** („*) *[real,allocatable]*
    - % **bplast** („*) *[real,allocatable]*
    - % **sizethetab** *[integer]*
    - % **bslast** („*) *[real,allocatable]*
    - % **to_arrays** *[to_arrays_t]*
    - % **nthetabs** *[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** *initialize_riterthetablocking_openmp()*, *finalize_riterthetablocking_openmp()*, *do_iteration_thetablocking_openmp()*

Needed modules

- *blocking* (*nfs()*): Module containing blocking information

- *dtb_mod* (*get_dtblm()*, *get_dh_dtblm()*): This module contains magnetic field stretching and advection terms plus a separate omega-effect. It is used for movie output....

- *nonlinear_lm_mod* (nonlinear_lm_t())

- *radial_data* (*n_r_cmb()*, *n_r_icb()*)

- *truncation* (*lm_max()*, *l_max()*, *n_theta_maxstr()*, *n_phi_maxstr()*, *nrp()*, *lmp_max_dtb()*, *lmp_max()*, *n_r_maxstr()*): This module defines the grid points and the truncation

- *radial_functions* (*or2()*, *orho1()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *leg_helper_mod* (leg_helper_t())

- *logic* (*l_anel()*, *l_rot_ma()*, *l_b_nl_cmb()*, *l_mag()*, *l_mag_kin()*, *l_conv()*, *l_conv_nl()*, *l_ht()*, *l_mag_nl()*, *l_cond_ic()*, *l_cond_ma()*, *l_mag_lf()*, *l_dtb()*, *l_movie_oc()*, *l_rot_ic()*, *l_store_frame()*, *l_heat()*, *l_b_nl_icb()*): Module containing the logicals that control the run

- *courant_mod* (*courant()*)

- *torsional_oscillations* (*gettonext()*, *gettofinish()*, *getto()*): This module contains information for TO calculation and output

- *outrot* (*get_lorentz_torque()*)

- *precision_mod*: This module controls the precision used in MagIC

- *riterthetablocking_mod* (riterthetablocking_t())

- *graphout_mod* (*graphout_mpi()*)

- *nonlinear_bcs* (*get_br_v_bcs()*)

- *out_movie* (*store_movie_frame()*)

- *nl_special_calc*: This module allows to calculcate several diagnostics that need to be computed in the physical space (non-linear quantities)

- *constants* (*zero()*): module containing constants and parameters used in the code.

- grid_space_arrays_mod (grid_space_arrays_t())

Types

- **type** riterthetablocking_openmp_mod/**unknown_type**

    Type fields

    – **% nl_lm** (*) *[nonlinear_lm_t,allocatable]*

    – **% nthreads** *[integer]*

    – **% lorentz_torque_ma** (*) *[real,allocatable]*

    – **% lorentz_torque_ic** (*) *[real,allocatable]*

    – **% gsa** (*) *[grid_space_arrays_t,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** *get_td()*, *set_zero()*, *finalize()*

Needed modules

- *blocking* (*lm2l()*, *lmp2lmpa()*, *lm2m()*, *lm2lmp()*, *lmp2lmps()*, *lm2lms()*, *lm2lma()*, *st_map()*): Module containing blocking information

- *horizontal_data* (*dtheta3s()*, *dtheta2a()*, *dtheta1a()*, *dtheta3a()*, *dtheta4a()*, *dlh()*, *dphi0()*, *dtheta1s()*, *dtheta2s()*, *dphi()*, *dtheta4s()*, *hdif_v()*, *hdif_b()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

- *fields* (*w_rloc()*, *dw_rloc()*, *z_rloc()*): This module contains the potential fields and their radial derivatives

- *truncation* (*lm_max()*, *l_max()*, *lm_maxmag()*): This module defines the grid points and the truncation

- *radial_functions* (*temp0()*, *or4()*, *or1()*, *beta()*, *r()*, *epscprof()*, *rgrav()*, *or2()*, *rho0()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *leg_helper_mod* (leg_helper_t())

- *rms* (*lfpol2hint()*, *cortor2hint()*, *lfpolas2hint()*, *maglmr()*, *cortoras2hint()*, *buolmr()*, *magas2hint()*, *geolmr()*, *lftor2hint()*, *advpol2hint()*, *arclmr()*, *arc2hint()*, *arcas2hint()*, *buo2hint()*, *advtor2hint()*, *geoas2hint()*, *advpollmr()*, *buoas2hint()*, *corpollmr()*, *advpolas2hint()*, *lftoras2hint()*, *pre2hint()*, *preas2hint()*, *mag2hint()*, *prelmr()*, *lfpollmr()*, *corpolas2hint()*, *advtoras2hint()*, *corpol2hint()*, *geo2hint()*): This module contains the global array used when RMS force balance is requested

- *precision_mod*: This module controls the precision used in MagIC

- *physical_parameters* (*ra()*, *n_r_lcr()*, *corfac()*, *ohmlossfac()*, *vischeatfac()*, *epsc()*): Module containing the physical parameters

- *rms_helpers* (*hint2pol()*, *hint2tor()*): This module contains several useful subroutines required to compute RMS diagnostics

- *logic* (*l_anel()*, *l_anelastic_liquid()*, *l_mag()*, *l_mag_kin()*, *l_conv()*, *l_conv_nl()*, *l_mag_lf()*, *l_mag_nl()*, *l_heat()*, *l_corr()*, *l_rmstest()*): Module containing the logicals that control the run

- *constants* (*two()*, *zero()*): module containing constants and parameters used in the code.

Types

- **type** `nonlinear_lm_mod`/**unknown_type**

    Type fields

    - **%** **vischeatlm** (*) *[complex,allocatable]*

    - **%** **lfplm** (*) *[complex,allocatable]*

    - **%** **advplm** (*) *[complex,allocatable]*

    - **%** **vxbplm** (*) *[complex,allocatable]*

    - **%** **lfrlm** (*) *[complex,allocatable]*

    - **%** **lftlm** (*) *[complex,allocatable]*

    - **%** **vxbtlm** (*) *[complex,allocatable]*

    - **%** **advrlm** (*) *[complex,allocatable]*

    - **%** **vsplm** (*) *[complex,allocatable]*

    - **%** **vxbrlm** (*) *[complex,allocatable]*

    - **%** **ohmlosslm** (*) *[complex,allocatable]*

    - **%** **vstlm** (*) *[complex,allocatable]*

    - **%** **vsrlm** (*) *[complex,allocatable]*

    - **%** **advtlm** (*) *[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 dVS-rLM 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 `hint2pol()`, `hint2tor()`

10.8.7 `nonlinear_bcs.f90`

Quick access

Routines `get_b_nl_bcs()`, `get_br_v_bcs()`, `v_rigid_boundary()`

Needed modules

- `blocking` (`lm2l()`, `sizethetab()`, `lmp2lmpa()`, `lm2m()`, `nfs()`, `lm2lmp()`, `lmp2lmps()`):
  Module containing blocking information
- `legendre_grid_to_spec` (`legtf2()`)
- `precision_mod`: This module controls the precision used in MagIC
- `physical_parameters` (`prmag()`, `sigma_ratio()`, `conductance_ma()`): Module containing the physical parameters
- `radial_data` (`n_r_cmb()`, `n_r_icb()`)
- `truncation` (`n_phi_max()`, `lmp_max()`, `nrp()`): This module defines the grid points and the truncation
- `radial_functions` (`r_cmb()`, `rho0()`, `r_icb()`): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- `horizontal_data` (`dtheta1s()`, `dtheta1a()`, `dphi()`, `sn2()`, `dlh()`, `o_sin_theta()`, `costheta()`): 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
- `constants` (`two()`): module containing constants and parameters used in the code.

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_seq()`, *do_iteration_thetablocking_openmp()*

>   **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*)

10.8. Non-linear part of the time stepping (radial loop)

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()*

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_even()*, *get_chebs_recurr()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC

- *logic* (*l_newmap()*): Module containing the logicals that control the run

- *constants* (*two()*, *pi()*, *one()*, *four()*, *half()*): module containing constants and parameters used in the code.

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()`*

10.9. Chebyshev polynomials and cosine transforms 231

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()*, *radial()*, *init_rnb()*

10.9.2 `init_costf.f90`

Quick access

> **Routines** *init_costf2()*, *init_costf1()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *useful* (*factorise()*): library with several useful subroutines
- *constants* (*two()*, *sin60()*, *cos72()*, *sin36()*, *cos36()*, *half()*, *pi()*, *sin72()*, *one()*): module containing constants and parameters used in the code.

Variables

Subroutines and functions

subroutine init_costf/**init_costf1** (*n*, *i_costf_init*, *ni*, *d_costf_init*, *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**
>
> - **n** *[integer,in]* :: No of grid points !
> - **i_costf_init** (*ni*) *[integer,out]* :: array for integers
> - **ni** *[integer,in]* :: dimension of i_costf_init
> - **d_costf_init** (*nd*) *[real,out]* :: array for integers
> - **nd** *[integer,in]* :: dimension of d_costf_init
>
> Called from *chebintinit()*, *mapdatar()*, *radial()*, *init_rnb()*
>
> Call to *factorise()*

subroutine init_costf/**init_costf2** (*n*, *i_costf_init*, *ni*, *d_costf_init*, *nd*)

> Purpose of this subroutine is to calculate several things needed for the cheb transform. Prepares costf2 for even number of grid points.
>
> **Parameters**
>
> - **n** *[integer,in]* :: No of grid points !
> - **i_costf_init** (*ni*) *[integer,out]* :: array for integers
> - **ni** *[integer,in]* :: dimension of i_costf_init
> - **d_costf_init** (*nd*) *[real,out]* :: array for integers
> - **nd** *[integer,in]* :: dimension of i_costf_init

Called from *radial()*

Call to *factorise()*

10.9.3 `cosine_transform.f90`

Quick access

> **Variables** *costf1*
>
> **Routines** *costf1_real()*, *costf2()*, *costf1_complex()*, *costf1_complex_1d()*,
> *costf1_real_1d()*

Needed modules

- *truncation* (*lm_max()*): This module defines the grid points and the truncation
- *precision_mod*: This module controls the precision used in MagIC
- *fft_fac_mod* (*fft_fac_real()*, *fft_fac_complex()*)
- *constants* (*two()*, *one()*, *half()*): module containing constants and parameters used in the code.

Variables

- cosine_transform/**costf1** *[public]*

Subroutines and functions

subroutine cosine_transform/**costf1_complex** (*f*, *n_f_max*, *n_f_start*, *n_f_stop*, *f2*, *i_costf_init*,
d_costf_init)

> 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**
>
> - **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
> - **i_costf_init** (*) *[integer,in]* :: prestored integers
> - **d_costf_init** (*) *[real,in]* :: prestored 1 numbers
>
> **Call to** *fft_fac_complex()*

subroutine cosine_transform/**costf1_complex_1d** (*f*, *f2*, *i_costf_init*, *d_costf_init*)

> **Parameters**

- **f** (*) *[complex,inout]* :: data/coeff input

- **f2** (*) *[complex,out]* :: work array of the same size as f

- **i_costf_init** (*) *[integer,in]* :: prestored integers

- **d_costf_init** (*) *[real,in]* :: prestored 1 numbers

Call to *fft_fac_complex()*

subroutine cosine_transform/**costf1_real** (*f*, *n_f_max*, *n_f_start*, *n_f_stop*, *f2*, *i_costf_init*, *d_costf_init*)

Parameters

- **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

- **i_costf_init** (*) *[integer,in]* :: prestored integers

- **d_costf_init** (*) *[real,in]* :: prestored 1 numbers

Use *truncation* (*lm_max_real()*)

Call to *fft_fac_real()*

subroutine cosine_transform/**costf1_real_1d** (*f*, *f2*, *i_costf_init*, *d_costf_init*)

Parameters

- **f** (*) *[real,inout]* :: data/coeff input

- **f2** (*) *[real,out]* :: work array of the same size as f

- **i_costf_init** (*) *[integer,in]* :: prestored integers

- **d_costf_init** (*) *[real,in]* :: prestored 1 numbers

Call to *fft_fac_real()*

subroutine cosine_transform/**costf2** (*f*, *n_f_max*, *n_f_start*, *n_f_stop*, *f2*, *i_costf_init*, *d_costf_init*, *isign*)

Parameters

- **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

- **i_costf_init** (*) *[integer,in]* :: prestored integers

- **d_costf_init** (*) *[real,in]* :: prestored 1 numbers

- **isign** *[integer,in]* :: = +1 (-1) for forward (backward) transform

Called from *get_drns_even()*, *get_ddrns_even()*, *get_ddr_even()*

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* (*sin36()*, *sin72()*, *sin60()*, *cos36()*, *cos72()*): 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()*, *costf1_real_1d()*

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 *costf2()*, *costf1_complex()*, *costf1_complex_1d()*

10.10 Legendre transforms

10.10.1 `plms.f90`

Quick access

Routines *plm_thetaas()*, *plm_theta()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC

- *constants* (*two()*, *osq4pi()*, *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 *outto()*, *outpv()*, *getegeos()*, *lnpas2tr()*, *horizontal()*

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 *legprep_ic()*, *legprep()*, *legprepg()*, *initialize()*

Needed modules

- *grenoble* (*db0()*, *lgrenoble()*, *b0()*, *ddb0()*): This module contains all variables for the case of an imposed IC dipole
- *blocking* (*lm2l()*, *lm2()*, *lm2m()*): Module containing blocking information
- *constants* (*two()*, *one()*, *zero()*): module containing constants and parameters used in the code.
- *precision_mod*: This module controls the precision used in MagIC
- *logic* (*l_fluxprofs()*, *l_mag_lf()*, *l_mag()*, *l_mag_kin()*, *l_conv()*, *l_movie_oc()*, *l_heat()*): Module containing the logicals that control the run
- *radial_data* (*n_r_cmb()*, *n_r_icb()*)
- *truncation* (*lm_max()*, *l_max()*): This module defines the grid points and the truncation
- *radial_functions* (*or2()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *horizontal_data* (*dLh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *fields* (*omega_ma()*, *omega_ic()*, *dj_rloc()*, *aj_rloc()*, *dw_rloc()*, *z_rloc()*, *ddw_rloc()*, *db_rloc()*, *w_rloc()*, *ddb_rloc()*, *p_rloc()*, *b_rloc()*, *dp_rloc()*, *dz_rloc()*, *ds_rloc()*, *s_rloc()*): This module contains the potential fields and their radial derivatives
- *torsional_oscillations* (*ddzASL()*): This module contains information for TO calculation and output

Types

- **type** `leg_helper_mod`/**unknown_type**

        Type fields

            – **% dzas** (*) *[real,allocatable]*

            – **% sr** (*) *[complex,allocatable]*

            – **% omegaic** *[real]*

            – **% zas** (*) *[real,allocatable]*

            – **% dsr** (*) *[complex,allocatable]*

            – **% dvhdrc** (*) *[complex,allocatable]*

            – **% dpr** (*) *[complex,allocatable]*

            – **% dlhb** (*) *[complex,allocatable]*

            – **% dlhz** (*) *[complex,allocatable]*

            – **% bhc** (*) *[complex,allocatable]*

            – **% cbhg** (*) *[complex,allocatable]*

            – **% prer** (*) *[complex,allocatable]*

            – **% dlhdw** (*) *[complex,allocatable]*

            – **% bcmb** (*) *[complex,allocatable]*

            – **% dlhj** (*) *[complex,allocatable]*

            – **% dlhw** (*) *[complex,allocatable]*

            – **% ddzas** (*) *[real,allocatable]*

            – **% cbhc** (*) *[complex,allocatable]*

            – **% bhg** (*) *[complex,allocatable]*

            – **% vhg** (*) *[complex,allocatable]*

            – **% omegama** *[real]*

            – **% vhc** (*) *[complex,allocatable]*

            – **% dvhdrg** (*) *[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*, *l_frame*, *ltonext*, *ltonext2*, *lto-calc*)

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 ?
- **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*, *dlh*, *lm_max*, *l_max*, *minc*, *r*, *lderiv*, *lhor*, *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 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()*, *legtfg()*, *legtf()*, *legtfgnomag()*

Needed modules

- *blocking* (*nfs()*, *sizethetab()*, *lm2mc()*, *lm2()*): Module containing blocking information

- *precision_mod*: This module controls the precision used in MagIC

- *parallel_mod* (rank()): This module contains the blocking information

- *truncation* (*lm_max()*, *l_max()*, *nrp()*, *n_m_max()*): This module defines the grid points and the truncation

- *logic* (*l_ht()*, *l_heat()*): Module containing the logicals that control the run

- *leg_helper_mod* (leg_helper_t())

- *horizontal_data* (*lstart()*, *osn2()*, *lmodd()*, *lstop()*, *plm()*, *dplm()*, *d_mc2m()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

- *constants* (*half()*, *one()*, *zero()*): module containing constants and parameters used in the code.

Variables

Subroutines and functions

subroutine legendre_spec_to_grid/**legtfg**(*nbc*, *lderiv*, *lviscbccalc*, *lfluxprofcalc*, *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]*
- **lfluxprofcalc** *[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, lfluxprofcalc, nthetas-tart, 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]*
- **lfluxprofcalc** *[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**(*dlhw, vhg, vhc, dlhz, cvhg, cvhc, l_max, minc, nthetas-tart, 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

- **dlhw** (*) *[complex,in]*
- **vhg** (*) *[complex,in]*
- **vhc** (*) *[complex,in]*
- **dlhz** (*) *[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 *store_movie_frame_ic()*, *graphout_ic()*, *fields_average()*

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 *outmisc()*, *outpar()*, *outperppar()*

10.10.4 `legendre_grid_to_spec.f90`

Quick access

> **Routines** *legtfas()*, *legtf2()*, *legtf1()*, *legtfas2()*, *legtf3()*

Needed modules

- *truncation* (*lmp_max()*, *n_theta_max()*, *nrp()*, *n_m_max()*): This module defines the grid points and the truncation

- *blocking* (*nfs()*, *sizethetab()*): Module containing blocking information

- *precision_mod*: This module controls the precision used in MagIC

- *horizontal_data* (*lstartp()*, *lstopp()*, *wplm()*, *lmoddp()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

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 *[inits()](), [transform_to_lm_space()](), [initv()]()*

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 *[get_dtblm()](), [transform_to_lm_space()](), [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 *[get_dtblm()](), [transform_to_lm_space()]()*

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 *[outto()](), [get_fluxes()](), [get_nlblayers()]()*

---

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 `outto()`, `get_fluxes()`, `get_perppar()`, `get_nlblayers()`, `getto()`, `get_helicity()`

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 `ni`, `d_fft_init`, `i_fft_init`, `nd`

Routines `wpass3jw()`, `wpass5jw()`, `init_fft()`, `fft_thetab()`, `wpass2jw()`, `fft99bjw()`, `fftjw()`, `fft_to_real()`, `fft99ajw()`, `wpass4jw()`

Needed modules

- `blocking`: Module containing blocking information
- `precision_mod`: This module controls the precision used in MagIC
- `useful` (`factorise()`): library with several useful subroutines
- `parallel_mod`: This module contains the blocking information
- `truncation`: This module defines the grid points and the truncation
- `constants` (`two()`, `sin60()`, `cos72()`, `sin36()`, `sin72()`, `half()`, `pi()`, `cos36()`, `one()`): module containing constants and parameters used in the code.

Variables

- fft/**i_fft_init** (100) *[integer,private]*

- fft/**ni** *[integer,private/parameter=100]*

- fft/**d_fft_init** (:) *[real,private/allocatable]*

- fft/**nd** *[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** *getpvptr()*, *getdvptr()*
>
> **Call to** *fftjw()*

subroutine fft/**fft_thetab** (*f*, *dir*)

> **Parameters**
>
> > - **f** (*nrp*,*nfs*) *[real,inout]*
> >
> > - **dir** *[integer,in]* :: back or forth transform
>
> **Called from** *store_movie_frame_ic()*, *get_btor()*, *graphout_ic()*, *get_dtblm()*, *inits()*, *transform_to_lm_space()*, *lm2pt()*, *initv()*, *get_br_v_bcs()*, *get_b_surface()*, *get_bpol()*, *transform_to_grid_space()*, *fields_average()*
>
> **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 $x(j)=\text{sum}(k=0,...,n-1)(c(k)*\exp(2*i*j*k*pi/n))$ where $c(k)=a(k)+i*b(k)$ and $c(n-k)=a(k)-i*b(k)$ and $i=\text{sqrt}(-1)$ 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 $c(k)=(1/n)*\text{sum}(j=0,...,n-1)(x(j)*\exp(-2*i*j*k*pi/n))$ where $c(k)=a(k)+i*b(k)$ and $i=\text{sqrt}(-1)$ 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()*, *cgesl()*, *sgefa()*, *cgeslml()*

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** *updatez()*, *j_cond()*

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()*, *updates()*, *updatewp()*, *updateb()*, *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 *updates()*, *inits()*, *s_cond()*, *getbackground()*, *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 *inits()*, *get_zmat()*, *s_cond()*, *get_s0mat()*, *getbackground()*, *get_z10mat()*, *get_smat()*, *get_wpmat()*, *j_cond()*, *get_bmat()*

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_dcheb_complex()*, *get_dddcheb()*, *get_ddcheb()*, *get_ddr()*, *get_dcheb_real_1d()*, *get_dr_complex()*, *get_dr_real_1d()*, *get_drns()*, *get_dddr()*

Needed modules

- *cosine_transform* (*costf1()*)
- *precision_mod*: This module controls the precision used in MagIC
- *constants* (*three()*, *one()*, *zero()*): 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*, *i_costf_init*, *d_costf_init*, *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,*) *[complex,in]*
> - **df** (n_f_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,*) *[complex,out]* :: work array needed for costf

- **work2** (n_f_max,*n_r_max*) *[complex,out]* :: work array for f transfer

- **i_costf_init** (*) *[integer,in]* :: info for costf

- **d_costf_init** (*) *[real,in]* :: info for costf

- **drx** (*) *[real,in]* :: first derivatives of x(r)

subroutine radial_der/**get_dr_real_1d**(*f*, *df*, *n_r_max*, *n_cheb_max*, *work1*, *work2*, *i_costf_init*, *d_costf_init*, *drx*)

> ### Parameters
>
> - **f** (*) *[real,in]*
>
> - **df** (*) *[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** (*) *[real,out]* :: work array needed for costf
>
> - **work2** (*n_r_max*) *[real,out]* :: work array for f transfer
>
> - **i_costf_init** (*) *[integer,in]* :: info for costf
>
> - **d_costf_init** (*) *[real,in]* :: info for costf
>
> - **drx** (*) *[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*, *i_costf_init*, *d_costf_init*, *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,*) *[complex,inout]*
>
> - **df** (n_f_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,*) *[complex,out]* :: work array needed for costf
>
> - **i_costf_init** (*) *[integer,in]* :: info for costf
>
> - **d_costf_init** (*) *[real,in]* :: info for costf
>
> - **drx** (*) *[real,in]* :: first derivatives of x(r)
>
> **Called from** *updates()*, *dtvrms()*, *write_dtb_frame()*, *dtbrms()*, *updateb()*, *fields_average()*, *get_dtblmfinish()*, *updates_ala()*

---

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*, *i_costf_init*, *d_costf_init*, *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,*) *[complex,in]*
- **df** (n_f_max,*) *[complex,out]* :: first derivative of f
- **ddf** (n_f_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,*) *[complex,out]* :: work array needed for costf
- **work2** (n_f_max,*) *[complex,out]* :: work array for f transfer
- **i_costf_init** (*) *[integer,in]* :: info for costf
- **d_costf_init** (*) *[real,in]* :: info for costf
- **drx** (*) *[real,in]* :: first derivatives of x(r)
- **ddrx** (*) *[real,in]* :: second derivatives of x(r)

Called from *updatez()*,   *getstartfields()*,   *updates()*,   *updateb()*, *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*, *i_costf_init*, *d_costf_init*, *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,*) *[complex,in]*
- **df** (n_f_max,*) *[complex,out]* :: first derivative of f
- **ddf** (n_f_max,*) *[complex,out]* :: second derivative of f
- **dddf** (n_f_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,*) *[complex,out]* :: work array needed for costf

- **work2** (n_f_max,*) *[complex,out]* :: work array needed for costf

- **i_costf_init** (*) *[integer,in]* :: info for costf

- **d_costf_init** (*) *[real,in]* :: info for costf

- **drx** (*) *[real,in]* :: first derivatives of x(r)

- **ddrx** (*) *[real,in]* :: second derivatives of x(r)

- **dddrx** (*) *[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,*) *[complex,in]*

- **df** (n_f_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** (*) *[real,in]*

- **df** (*) *[real,out]*

- **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_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,*) *[complex,in]*

- **df** (n_f_max,*) *[complex,out]*

- **ddf** (n_f_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,*) *[complex,in]*

- **df** (n_f_max,*) *[complex,out]*

- **ddf** (n_f_max,*) *[complex,out]*

- **dddf** (n_f_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_dcheb_even()*, *get_drns_even()*, *get_ddrns_even()*, *get_ddr_even()*

Needed modules

- *cosine_transform* (*costf2()*, *costf1()*)

- *precision_mod*: This module controls the precision used in MagIC

- *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*, *i_costf1_init*, *d_costf1_init*, *i_costf2_init*, *d_costf2_init*)

> 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,*) *[complex,in]*
> - **df** (n_f_max,*) *[complex,out]* :: first derivative of f
> - **ddf** (n_f_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,*) *[complex,out]* :: work array needed for costf
> - **work2** (n_f_max,*) *[complex,out]* :: work array needed for costf
> - **i_costf1_init** (*) *[integer,in]*
> - **d_costf1_init** (*) *[real,in]*
> - **i_costf2_init** (*) *[integer,in]*
> - **d_costf2_init** (*) *[real,in]*

> **Called from** *getstartfields()*, *updateb()*

> **Call to** *get_ddcheb_even()*, *costf2()*

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*, *i_costf1_init*, *d_costf1_init*, *i_costf2_init*, *d_costf2_init*)

> 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,*) *[complex,inout]*

- **df** (n_f_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,*) *[complex,out]* :: work array needed for costf
- **i_costf1_init** (*) *[integer,in]*
- **d_costf1_init** (*) *[real,in]*
- **i_costf2_init** (*) *[integer,in]*
- **d_costf2_init** (*) *[real,in]*

Called from *write_dtb_frame()*, *fields_average()*

Call to *get_dcheb_even()*, *costf2()*

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*, *i_costf1_init*, *d_costf1_init*, *i_costf2_init*, *d_costf2_init*)

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,*) *[complex,inout]*
- **df** (n_f_max,*) *[complex,out]* :: first derivative of f
- **ddf** (n_f_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,*) *[complex,out]* :: work array needed for costf
- **i_costf1_init** (*) *[integer,in]*
- **d_costf1_init** (*) *[real,in]*
- **i_costf2_init** (*) *[integer,in]*
- **d_costf2_init** (*) *[real,in]*

Called from *fields_average()*

Call to *get_ddcheb_even()*, *costf2()*

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,*) *[complex,in]*
> > - **df** (n_f_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,*) *[complex,in]*
> > - **df** (n_f_max,*) *[complex,out]*
> > - **ddf** (n_f_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_ddrns_even()*, *get_ddr_even()*

10.13.3 `integration.f90`

Description

Radial integration functions

Quick access

> **Routines** *rint_r()*, *rint()*, *rintic()*

Needed modules

- *cosine_transform* (*costf1()*)
- *precision_mod*: This module controls the precision used in MagIC
- *constants* (*two()*, *one()*, *half()*): module containing constants and parameters used in the code.

Variables

Subroutines and functions

function integration/**rint** (*f*, *nrmax*, *drfac*, *i_costf_init*, *d_costf_init*)

This function performs the radial integral over a function f that is given on the appropriate nRmax radial Chebychev grid points. The arrays i_costf_init,d_costf_init are defined by calling init_costf1.

Note: drFac maps radius to cheb space [-1,1] drFac=two/(rMax-rMin)

Parameters

- **f** (*) *[real,in]*
- **nrmax** *[integer,in]*
- **drfac** *[real,in]*
- **i_costf_init** (*) *[integer,in]*
- **d_costf_init** (*) *[real,in]*

Return **rint** *[real]*

Called from *outmisc()*, *spectrum_temp()*, *spectrum_temp_average()*, *outperppar()*

function integration/**rintic** (*f*, *nrmax*, *drfac*, *i_costf_init*, *d_costf_init*)

This function performs the radial integral over a function f that is given on the appropriate nRmax radial Chebychev grid points. The arrays i_costf_init,d_costf_init are defined by calling init_costf1.

Parameters

- **f** (*) *[real,inout]*
- **nrmax** *[integer,in]*
- **drfac** *[real,in]*
- **i_costf_init** (*) *[integer,in]*
- **d_costf_init** (*) *[real,in]*

Return **rintic** *[real]*

Called from *get_power()*, *spectrum()*, *get_e_mag()*

function integration/**rint_r** (*f*, *n_r_max*, *n_cheb_max*, *dr_fac*, *i_costf_init*, *d_costf_init*)

Parameters

- **f** (*) *[real,in]*

- **n_r_max** *[integer,in]*

- **n_cheb_max** *[integer,in]*

- **dr_fac** (*) *[real,in]*

- **i_costf_init** (*) *[integer,in]*

- **d_costf_init** (*) *[real,in]*

Return **rint_r** *[real]*

Called from *outto()*, *spectrum_average()*, *output()*, *get_u_square()*, *getdlm()*, *dtvrms()*, *get_power()*, *get_poltorrms()*, *precalc()*, *dtbrms()*, *spectrum()*, *get_angular_moment()*, *get_e_mag()*, *get_e_kin()*

10.14 Blocking and LM mapping

10.14.1 `blocking.f90`

Description

Module containing blocking information

Quick access

Variables *cacheblock_size_in_b*, *sizethetabi*, *nbsave*, *nbdown*, *lm22m*, *sizelmb2*, *lmp2*, *lm22lm*, *lm2m*, *lmp2lmpa*, *lmp2l*, *lmstartb*, *l2lmas*, *lm2lms*, *lmp2lmps*, *lm2lma*, *lm2lmp*, *lmp2lm*, *lm2*, *lm22l*, *lmstopb*, *lm2l*, *nlmbs2*, *lm2mc*, *get_theta_blocking*, *st_sub_map*, *lo_sub_map*, *st_map*, *sn_sub_map*, *lo_map*, *nfs*, *sizelmb*, *nlmbs*, *nthetabs*, *sizerb*, *sizethetab*

Routines *get_standard_lm_blocking()*, *get_lorder_lm_blocking()*, *get_subblocks()*, *get_theta_blocking_cache()*, *get_theta_blocking_openmp()*, *get_snake_lm_blocking()*, *initialize_blocking()*

Needed modules

- *output_data* (*log_file()*, *nlf()*): This module contains the parameters for output control

- *precision_mod*: This module controls the precision used in MagIC

- *parallel_mod* (*n_procs()*, *nlmbs_per_rank()*, *nthreads()*, rank(), *rank_with_l1m0()*): This module contains the blocking information

- *truncation* (*n_theta_max()*, *lm_max()*, *l_max()*, *nrp()*, *minc()*, *lmp_max()*, *n_r_max()*): This module defines the grid points and the truncation

- *lmmapping* (*allocate_subblocks_mappings()*, mappings(), *allocate_mappings()*, subblocks_mappings())

- *logic* (*l_save_out()*): Module containing the logicals that control the run

- *useful* (*logwrite()*): library with several useful subroutines

Variables

- blocking/**lm22m** (:,:,:,:) *[integer,pointer/public]*

- blocking/**nlmbs2** (:) *[integer,pointer/public]*

- blocking/**lmstopb** (:) *[integer,allocatable/public]*

- blocking/**lmstartb** (:) *[integer,allocatable/public]*

- blocking/**l2lmas** (:) *[integer,pointer/public]*

- blocking/**sn_sub_map** *[subblocks_mappings,target/public]*

- blocking/**lmp2lmps** (:) *[integer,pointer/public]*

- blocking/**lo_sub_map** *[subblocks_mappings,target/public]*

- blocking/**nbsave** *[integer,private/parameter=16]*

- blocking/**lmp2lmpa** (:) *[integer,pointer/public]*

- blocking/**sizelmb** *[integer,public]*

- blocking/**sizethetabi** *[integer,private/parameter=284]*

- blocking/**lm2mc** (:) *[integer,pointer/public]*

- blocking/**lm22l** (:,:,:,:) *[integer,pointer/public]*

- blocking/**lmp2lm** (:) *[integer,pointer/public]*

- blocking/**lmp2** (:,:) *[integer,pointer/public]*

- blocking/**cacheblock_size_in_b** *[integer,public]*

- blocking/**nthetabs** *[integer,public]*

- blocking/**lm22lm** (:,:,:,:) *[integer,pointer/public]*

- blocking/**lm2** (:,:) *[integer,pointer/public]*

- blocking/**nbdown** *[integer,private/parameter=8]*

- blocking/**sizerb** *[integer,public]*

- blocking/**sizethetab** *[integer,public]*

- blocking/**lm2lmp** (:) *[integer,pointer/public]*

- blocking/**lmp2l** (:) *[integer,pointer/public]*

- blocking/**lm2lma** (:) *[integer,pointer/public]*

- blocking/**lm2lms** (:) *[integer,pointer/public]*

- blocking/**lm2l** (:) *[integer,pointer/public]*

- blocking/**nfs** *[integer,public]*

- blocking/**nlmbs** *[integer,public]*

- blocking/**st_map** *[mappings,target/public]*

- blocking/**lo_map** *[mappings,target/public]*

- blocking/**lm2m** (:) *[integer,pointer/public]*

- blocking/**get_theta_blocking** *[public]*

- blocking/**st_sub_map** *[subblocks_mappings,target/public]*

- blocking/**sizelmb2** (:,:) *[integer,pointer/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

    - % **lmp_max** *[integer]*
    - % **lm2lmp** (*) *[integer,allocatable]*
    - % **lmp2l** (*) *[integer,allocatable]*
    - % **lm2lms** (*) *[integer,allocatable]*
    - % **lm2lma** (*) *[integer,allocatable]*
    - % **lm2l** (*) *[integer,allocatable]*
    - % **l2lmas** (*) *[integer,allocatable]*
    - % **lmp2lmps** (*) *[integer,allocatable]*
    - % **l_max** *[integer]*
    - % **lmp2lmpa** (*) *[integer,allocatable]*
    - % **lmp2lm** (*) *[integer,allocatable]*
    - % **lm2mc** (*) *[integer,allocatable]*
    - % **lm_max** *[integer]*
    - % **lmp2** (,) *[integer,allocatable]*
    - % **lm2m** (*) *[integer,allocatable]*
    - % **lm2** (,) *[integer,allocatable]*

- **type** lmmapping/**unknown_type**

Type fields

- **%** **lm22m** (,,*) *[integer,allocatable]*

- **%** **l_max** *[integer]*

- **%** **sizelmb2max** *[integer]*

- **%** **nlmbs2** (*) *[integer,allocatable]*

- **%** **lm22l** (,,*) *[integer,allocatable]*

- **%** **nlmbs** *[integer]*

- **%** **sizelmb2** (,) *[integer,allocatable]*

- **%** **lm22lm** (,,*) *[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*, *lm-stopb*)

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_t_r_sets*, *n_b_r_sets*, *n_v_r_sets*, *lbdissmean*, *dipcmbmean*, *rmmean*, *elmean*, *e_mag_tmean*, *dlvmean*, *dlvcmean*, *timenormlog*, *dmvmean*, *timepassedlog*, *dmbmean*, *dlbmean*, *geosmean*, *e_kin_pmean*, *dzvmean*, *e_mag_pmean*, *etot*, *dpvmean*, *dipmean*, *dteint*, *elcmbmean*, *lvdissmean*, *etotold*, *e_kin_tmean*, *rolmean*, *ntpotsets*, *nrms_sets*, *n_dt_cmb_sets*, *n_cmb_setsmov*, *nlogs*, *ntosets*, *nbpotsets*, *n_e_sets*, *ntormssets*, *ntomovsets*, *nvpotsets*, *n_spec*, *npvsets*

Routines *output()*, *initialize_output()*

Needed modules

- *blocking* (*lo_map()*, *st_map()*, *lm2()*): Module containing blocking information

- *communications* (*gt_ic()*, *gt_oc()*, *gather_all_from_lo_to_rank0()*)

- *num_param* (*tscale()*): Module containing numerical and control parameters

- *spectra* (*spectrum_temp_average()*, *spectrum_average()*, *spectrum()*, *spectrum_temp()*)

- *physical_parameters* (*prmag()*, *ktopv()*, *lffac()*, *nvarcond()*, *ek()*, *opm()*): Module containing the physical parameters

- *outpv3* (*outpv()*)

- *radial_data* (*nrstart()*, *n_r_cmb()*, *nrstartmag()*, *nrstopmag()*, *nrstop()*)

- *dtb_mod* (*get_dtblmfinish()*): This module contains magnetic field stretching and advection terms plus a separate omega-effect. It is used for movie output....

- *truncation* (*n_r_max()*, *l_max()*, *n_r_maxmag()*, *minc()*, *l_maxmag()*, *n_r_ic_max()*, *lm_max()*): This module defines the grid points and the truncation

- *outmisc_mod* (*outmisc()*)

- *movie_data* (*movie_gather_frames_to_rank0()*)

- *out_coeff* (*write_bcmb()*, *write_coeff_r()*): This module contains the subroutines that calculate the Bcmb files and the [B|V|T]_coeff_r files

- *out_movie* (*write_movie_frame()*)

- *radial_spectra*

- *kinetic_energy* (*get_e_kin()*, *get_u_square()*)

- *rms* (*zerorms()*): This module contains the global array used when RMS force balance is requested

- *out_rms* (*dtbrms()*, *dtvrms()*)

- *lmloop_data* (*llm()*, *ulmmag()*, *llmmag()*, *lm_per_rank()*, *ulm()*, *lm_on_last_rank()*)

- *charmanip* (*dble2str()*): This module contains several useful routines to manipule character strings

- *out_movie_ic* (*store_movie_frame_ic()*)

- *graphout_mod* (*graphout_ic()*)

- *magnetic_energy* (*get_e_mag()*)

- *storecheckpoints*: This module contains several subroutines that can be used to store the rst_#.TAG files

- *fields* (*dj_ic_lmloc()*, *z()*, *b_ic()*, *dw()*, *dj()*, *ddj_lmloc()*, *b()*, *z_lmloc()*, *omega_ic()*, *ds_lmloc()*, *ds()*, *aj_ic()*, *ddj()*, *w()*, *ddb_lmloc()*, *ddb_ic_lmloc()*, *db_lmloc()*, *aj_lmloc()*, *ddw_lmloc()*, *b_ic_lmloc()*, *db_ic_lmloc()*, *db_ic()*, *ddj_ic()*, *s_lmloc()*, *p()*, *ddb_ic()*, *dz_lmloc()*, *ddb()*, *ddw()*, *dz()*, *p_lmloc()*, *dw_lmloc()*, *ddj_ic_lmloc()*, *w_lmloc()*, *omega_ma()*, *s()*, *dj_ic()*, *dj_lmloc()*, *aj_ic_lmloc()*, *aj()*, *db()*, *b_lmloc()*): This module contains the potential fields and their radial derivatives

- *outpar_mod* (*outpar()*, *outperppar()*)

- *output_data* (*n_cmbmov_file()*, *log_file()*, *l_max_cmb()*, *n_par_file()*, *nlf()*, *b_r_file()*, *n_coeff_r_max()*, *tag()*, *n_r_array()*, *n_b_r_file()*, *n_r_step()*, *n_v_r_file()*, *cmb_file()*, *cmbmov_file()*, *dt_cmb_file()*, *par_file()*, *l_max_r()*, *rst_file()*, *n_t_r_file()*, *n_cmb_file()*, *t_r_file()*, *n_dt_cmb_file()*, *n_rst_file()*, *n_coeff_r()*, *v_r_file()*): This module contains the parameters for output control

- *radial_functions* (*r_cmb()*, *or2()*, *drx()*, *r()*, *i_costf_init()*, *r_icb()*, *or1()*, *d_costf_init()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *logic* (*l_anel()*, *l_mag()*, *l_perppar()*, *l_average()*, *l_power()*, *l_rmagspec()*, *l_sric()*, *l_save_out()*, *l_cond_ic()*, *l_dt_cmb_field()*, *l_non_rot()*, *l_r_fieldt()*, *l_r_field()*, *l_mag_lf()*, *l_dtb()*, *l_rms()*, *lverbose()*, *l_pv()*, *l_store_frame()*, *l_movie_ic()*, *l_cmb_field()*): Module containing the logicals that control the run

- *fieldslast* (*dzdtlast()*, *dwdtlast_lmloc()*, *dbdt_iclast_lmloc()*, *djdt_iclast()*, *djdtlast_lmloc()*, *dpdtlast()*, *dpdtlast_lmloc()*, *dbdtlast()*, *dzdtlast_lo()*, *dbdt_iclast()*, *djdtlast()*, *dsdtlast()*, *djdt_iclast_lmloc()*, *dwdtlast()*, *dbdtlast_lmloc()*, *dsdtlast_lmloc()*): This module contains time-derivaties array of the previous time-step They are needed in the time-stepping scheme.

- *useful* (*safeopen()*, *logwrite()*, *safeclose()*): library with several useful subroutines

- *omega* (*outomega()*): This module allows to compute the axisymmetric zonal flow versus the cylindrical radius s. By

- *outrot* (*write_rot()*)

- *outto_mod* (*outto()*)

- *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)

- *horizontal_data* (*dlh()*, *dpl0eq()*, *hdif_b()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

- *integration* (*rint_r()*): Radial integration functions

- *parallel_mod* (rank()): This module contains the blocking information

- *precision_mod*: This module controls the precision used in MagIC

- *fields_average_mod* (*fields_average()*): This module is used when one wants to store time-averaged quantities

- *getdlm_mod* (*getdlm()*)

- *constants* (*vol_ic()*, *two()*, *vol_oc()*, *mass()*, *surf_cmb()*): module containing constants and parameters used in the code.

- *power* (*get_power()*)

Variables

- output_mod/**n_dt_cmb_sets** *[integer,private]*

- output_mod/**nrms_sets** *[integer,private]*

- output_mod/**e_kin_tmean** *[real,private]*

- output_mod/**etot** *[real,private]*

- output_mod/**etotold** *[real,private]*

- output_mod/**rmmean** *[real,private/save]*

- output_mod/**n_spec** *[integer,private]*

- output_mod/**n_e_sets** *[integer,private]*

- output_mod/**ntpotsets** *[integer,private]*

- output_mod/**dlvmean** *[real,private/save]*

- output_mod/**lvdissmean** *[real,private/save]*

- output_mod/**n_v_r_sets** (:) *[integer,private/allocatable]*

- output_mod/**dlbmean** *[real,private/save]*

- output_mod/**geosmean** *[real,private/save]*

- output_mod/**ntomovsets** *[integer,private]*

- output_mod/**n_cmb_setsmov** *[integer,private]*

- output_mod/**nlogs** *[integer,private]*

- output_mod/**lbdissmean** *[real,private/save]*

- output_mod/**ntosets** *[integer,private]*

- output_mod/**dlvcmean** *[real,private/save]*

- output_mod/**elmean** *[real,private/save]*

- output_mod/**e_mag_pmean** *[real,private]*

- output_mod/**timepassedlog** *[real,private]*

- output_mod/**dteint** *[real,private]*

- output_mod/**e_mag_tmean** *[real,private]*

- output_mod/**dpvmean** *[real,private/save]*

- output_mod/**n_t_r_sets** (:) *[integer,private/allocatable]*

- output_mod/**dipcmbmean** *[real,private/save]*

- output_mod/**npvsets** *[integer,private]*

- output_mod/**dmbmean** *[real,private/save]*

- output_mod/**elcmbmean** *[real,private/save]*

- output_mod/**dipmean** *[real,private/save]*

- output_mod/**e_kin_pmean** *[real,private]*

- output_mod/**ntormssets** *[integer,private]*

- output_mod/**nbpotsets** *[integer,private]*

- output_mod/**nvpotsets** *[integer,private]*

- output_mod/**n_b_r_sets** (:) *[integer,private/allocatable]*

- output_mod/**rolmean** *[real,private/save]*

- output_mod/**dzvmean** *[real,private/save]*

- output_mod/**timenormlog** *[real,private]*

- output_mod/**dmvmean** *[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_u_square()*, *get_e_kin()*

Needed modules

- *blocking* (*lo_map()*, *st_map()*): Module containing blocking information
- *communications* (*get_global_sum()*)
- *num_param* (*escale()*, *tscale()*): Module containing numerical and control parameters
- *horizontal_data* (*dlh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *physical_parameters* (*ek()*, *prmag()*, *nvarcond()*): Module containing the physical parameters
- *output_data* (*tag()*, *u_square_file()*, *n_u_square_file()*, *n_e_kin_file()*, *e_kin_file()*): This module contains the parameters for output control
- *radial_functions* (*r_cmb()*, *sigma()*, *i_costf_init()*, *drx()*, *r()*, *d_costf_init()*, *orho2()*, *r_icb()*, *or2()*, *or1()*, *orho1()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *logic* (*l_save_out()*, *l_non_rot()*): Module containing the logicals that control the run
- *useful* (*cc2real()*): library with several useful subroutines
- *truncation* (*n_r_max()*, *l_max()*): This module defines the grid points and the truncation
- *precision_mod*: This module controls the precision used in MagIC
- *integration* (*rint_r()*): Radial integration functions
- *parallel_mod*: This module contains the blocking information
- *lmloop_data* (*llm()*, *ulm()*)
- *constants* (*two()*, *one()*, *four()*, *vol_oc()*, *pi()*, *half()*): module containing constants and parameters used in the code.

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** *output()*, *fields_average()*
>
> **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_ta*, *e_dipa*, *e_p_asa*, *e_t_asa*, *e_pa*

Routines *initialize_magnetic_energy()*, *get_e_mag()*

Needed modules

- *blocking* (*lmstartb()*, *lmstopb()*, *st_map()*, *lo_map()*): Module containing blocking information

- *bext* (*rrmp()*, *n_imp()*): Module containing the external field parameters

- *num_param* (*tscale()*, *escale()*): Module containing numerical and control parameters

- *horizontal_data* (*dlh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

- *parallel_mod*: This module contains the blocking information

- *radial_data* (*n_r_cmb()*)

- *truncation* (*n_r_ic_maxmag()*, *n_r_maxmag()*, *n_r_ic_max()*, *n_r_max()*): This module defines the grid points and the truncation

- *radial_functions* (*d_costf1_ic_init()*, *sigma()*, *drx()*, *dr_fac_ic()*, *orho1()*, *r_ic()*, *r_cmb()*, *i_costf_init()*, *r()*, *d_costf_init()*, *r_icb()*, *or2()*, *i_costf1_ic_init()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *logic* (*l_mag()*, *l_save_out()*, *l_mag_lf()*, *l_cond_ic()*): Module containing the logicals that control the run

- *useful* (*cc2real()*, *cc22real()*): library with several useful subroutines

- *output_data* (*tag()*, *n_dipole_file()*, *dipole_file()*, *e_mag_oc_file()*, *n_e_mag_oc_file()*, *n_e_mag_ic_file()*, *e_mag_ic_file()*): This module contains the parameters for output control

- *precision_mod*: This module controls the precision used in MagIC

- *integration* (*rint_r()*, *rintic()*): Radial integration functions

- *physical_parameters* (*lffac()*, *kbotb()*, *ktopb()*): Module containing the physical parameters

- *movie_data* (*moviedipstrengthgeo()*, *moviediplon()*, *moviedipstrength()*, *moviedipcolat()*)

- *lmloop_data* (*ulmmag()*, *llmmag()*)

- *constants* (*two()*, *one()*, *four()*, *zero()*, *pi()*, *half()*): module containing constants and parameters used in the code.

Variables

- magnetic_energy/**e_pa** (:) *[real,private/allocatable]*

- magnetic_energy/**e_ta** (:) *[real,private/allocatable]*

- magnetic_energy/**e_p_asa** (:) *[real,private/allocatable]*

- magnetic_energy/**e_dipa** (:) *[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 *output()*, *fields_average()*

        Call to *cc2real()*, *rint_r()*, *cc22real()*, *rintic()*

10.15.4 `getDlm.f90`

Quick access

>   **Routines** *getdlm()*

Needed modules

- *blocking* (*lo_map()*, *st_map()*): Module containing blocking information
- *integration* (*rint_r()*): Radial integration functions
- *num_param* (*escale()*): Module containing numerical and control parameters
- *precision_mod*: This module controls the precision used in MagIC
- *useful* (*cc2real()*, *cc22real()*): library with several useful subroutines
- *parallel_mod*: This module contains the blocking information
- *lmloop_data* (*llm()*, *ulm()*)
- *truncation* (*m_max()*, *minc()*, *l_max()*, *n_r_max()*): This module defines the grid points and the truncation
- *radial_functions* (*d_costf_init()*, *orho1()*, *or2()*, *drx()*, *i_costf_init()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *horizontal_data* (*dlh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *constants* (*pi()*, *half()*): module containing constants and parameters used in the code.

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

- *blocking* (*nfs()*, *sizethetab()*, *nthetabs()*, *lo_map()*): Module containing blocking information
- *num_param* (*lscale()*): Module containing numerical and control parameters
- *horizontal_data* (*gauss()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *parallel_mod*: This module contains the blocking information
- *radial_data* (*nrstart()*, *n_r_cmb()*, *n_r_icb()*, *nrstop()*)
- *output_data* (*tag()*, *n_misc_file()*, *misc_file()*): This module contains the parameters for output control
- *radial_functions* (*temp0()*, *botcond()*, *topcond()*, *kappa()*, *dtemp0()*, *r_cmb()*, *i_costf_init()*, *r()*, *d_costf_init()*, *r_icb()*, *dr_fac()*, *rho0()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *logic* (*l_heat()*, *l_hel()*, *l_anelastic_liquid()*, *l_save_out()*, *l_par()*): Module containing the logicals that control the run
- *legendre_spec_to_grid* (*lmas2pt()*)
- *useful* (*cc2real()*): library with several useful subroutines
- *truncation* (*n_r_max()*, *l_max()*, *lm_max()*): This module defines the grid points and the truncation
- *precision_mod*: This module controls the precision used in MagIC
- *integration* (*rint_r()*, *rint()*): Radial integration functions
- *physical_parameters* (*epss()*): Module containing the physical parameters
- *egeos_mod* (*getegeos()*)
- *lmloop_data* (*llm()*, *ulm()*)
- *constants* (*two()*, *osq4pi()*, *one()*, *four()*, *vol_oc()*, *pi()*, *sq4pi()*): module containing constants and parameters used in the code.

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()*,   *write_rot()*,   *get_lorentz_torque()*,
    get_viscous_torque_real(),   *get_angular_moment()*,
    get_viscous_torque_complex()

Needed modules

- *blocking* (*lmstartb()*, *lm2()*, *lmstopb()*, *lo_map()*, *st_map()*): Module containing blocking information

- *num_param* (*lscale()*, *tscale()*, *vscale()*): Module containing numerical and control parameters

- *horizontal_data* (*gauss()*, *costheta()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

- *parallel_mod*: This module contains the blocking information

- *radial_data* (*n_r_cmb()*, *n_r_icb()*)

- *output_data* (*tag()*, *n_angular_file()*, *n_srma_file()*, *rot_file()*, *n_rot_file()*, *n_sric_file()*, *srma_file()*, *sric_file()*, *angular_file()*): This module contains the parameters for output control

- *radial_functions* (*r_cmb()*, *i_costf_init()*, *drx()*, *r()*, *d_costf_init()*, *r_icb()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *logic* (*l_rot_ma()*, *l_am()*, *l_mag_lf()*, *l_save_out()*, *l_rot_ic()*, *l_mag()*, *l_sric()*, *l_srma()*, *l_drift()*, *l_iner()*): Module containing the logicals that control the run

- *grenoble* (*lgrenoble()*, *bic()*): This module contains all variables for the case of an imposed IC dipole

- *truncation* (*nrp()*, *n_r_max()*, *minc()*, *n_phi_max()*, *n_r_maxmag()*): This module defines the grid points and the truncation

- *precision_mod*: This module controls the precision used in MagIC

- *integration* (*rint_r()*, *rint()*): Radial integration functions

- *physical_parameters* (*ktopv()*, *kbotv()*): Module containing the physical parameters

- *lmloop_data* (*llm()*, *ulmmag()*, *llmmag()*, *ulm()*)

- *constants* (*two()*, *y11_norm()*, *y10_norm()*, *c_moi_ma()*, *third()*, *four()*, *half()*, *pi()*, *c_moi_oc()*, *c_moi_ic()*, *zero()*): module containing constants and parameters used in the code.

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_seq(), *do_iteration_thetablocking_openmp()*

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 *fresmeanr*, *eperpaximeanr*, *eparmeanr*, *mvar*, *eparaximeanr*, *dlvmeanr*, *fconvmeanr*, *gradt2meanr*, *svar*, *rolmeanr*, *eperpmeanr*, *dlvu2meanr*, *duhmeanr*, *uhmeanr*, *rmmeanr*, *fcondmeanr*, *dlvcmeanr*, *dlvu2cmeanr*, *fpoynmeanr*, *smeanr*, *rolmeanru2*, *fkinmeanr*, *fviscmeanr*

Routines *initialize_outpar_mod()*, *outpar()*, *outperppar()*

Needed modules

- *blocking* (*nfs()*, *sizethetab()*, *nthetabs()*, *lm2m()*): Module containing blocking information

- *num_param* (*tscale()*): Module containing numerical and control parameters

- *horizontal_data* (*gauss()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

- *physical_parameters* (*prmag()*, *kbots()*, *vischeatfac()*, *ek()*, *ohmlossfac()*, *ktops()*, *opr()*): Module containing the physical parameters

- *fields* (*ds_rloc()*, *s_rloc()*): This module contains the potential fields and their radial derivatives

- *radial_data* (*nrstart()*, *nrstartmag()*, *nrstopmag()*, *n_r_icb()*, *nrstop()*)

- *truncation* (*n_r_max()*, *l_max()*, *lm_max()*, *l_maxmag()*, *n_r_maxmag()*): This module defines the grid points and the truncation

- *radial_functions* (*temp0()*, *sigma()*, *i_costf_init()*, *r()*, *d_costf_init()*, *dr_fac()*, *or2()*, *kappa()*, *rho0()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *logic* (*l_anel()*, *l_fluxprofs()*, *l_viscbccalc()*, *l_save_out()*, *l_perppar()*, *l_mag_nl()*): Module containing the logicals that control the run

- *legendre_spec_to_grid* (*lmas2pt()*)

- *useful* (*cc2real()*): library with several useful subroutines

- *output_data* (*tag()*, *perppar_file()*, *n_perppar_file()*): This module contains the parameters for output control

- *precision_mod*: This module controls the precision used in MagIC

- *integration* (*rint()*): Radial integration functions

- *parallel_mod*: This module contains the blocking information

- *constants* (*two()*, *mass()*, *osq4pi()*, *four()*, *pi()*, *sq4pi()*, *half()*): module containing constants and parameters used in the code.

Variables

- outpar_mod/**eperpmeanr** (:) *[real,private/allocatable]*

- outpar_mod/**fcondmeanr** (:) *[real,private/allocatable]*

- outpar_mod/**uhmeanr** (:) *[real,private/allocatable]*

- outpar_mod/**rmmeanr** (:) *[real,private/allocatable]*

- outpar_mod/**dlvmeanr** (:) *[real,private/allocatable]*

- outpar_mod/**smeanr** (:) *[real,private/allocatable]*

- outpar_mod/**svar** (:) *[real,private/allocatable]*

- outpar_mod/**rolmeanr** (:) *[real,private/allocatable]*

- outpar_mod/**rolmeanru2** (:) *[real,private/allocatable]*

- outpar_mod/**fresmeanr** (:) *[real,private/allocatable]*

- outpar_mod/**fkinmeanr** (:) *[real,private/allocatable]*

- outpar_mod/**eparaximeanr** (:) *[real,private/allocatable]*

- outpar_mod/**eperpaximeanr** (:) *[real,private/allocatable]*

- outpar_mod/**eparmeanr** (:) *[real,private/allocatable]*

- outpar_mod/**dlvu2meanr** (:) *[real,private/allocatable]*

- outpar_mod/**dlvu2cmeanr** (:) *[real,private/allocatable]*

- outpar_mod/**fconvmeanr** (:) *[real,private/allocatable]*

- outpar_mod/**fviscmeanr** (:) *[real,private/allocatable]*

- outpar_mod/**gradt2meanr** (:) *[real,private/allocatable]*

- outpar_mod/**dlvcmeanr** (:) *[real,private/allocatable]*

- outpar_mod/**duhmeanr** (:) *[real,private/allocatable]*

- outpar_mod/**mvar** (:) *[real,private/allocatable]*

- outpar_mod/**fpoynmeanr** (:) *[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](n_r_max)*) *[real,in]* :: kinetic energy w radius
>>>
>>> - **rolru2** (*[n_r_max](n_r_max)*) *[real,in]*
>>>
>>> - **dlvr** (*[n_r_max](n_r_max)*) *[real,in]*
>>>
>>> - **dlvrc** (*[n_r_max](n_r_max)*) *[real,in]*
>>>
>>> - **dlvru2** (*[n_r_max](n_r_max)*) *[real,in]*
>>>
>>> - **dlvru2c** (*[n_r_max](n_r_max)*) *[real,in]*
>>>
>>> - **uhlmr** (*[l_max](l_max)*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
>>>
>>> - **duhlmr** (*[l_max](l_max)*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
>>>
>>> - **gradslmr** (*[l_max](l_max)*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
>>>
>>> - **fconvlmr** (*[l_max](l_max)*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
>>>
>>> - **fkinlmr** (*[l_max](l_max)*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
>>>
>>> - **fvisclmr** (*[l_max](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](n_r_max)*) *[real,out]*
>>
>> **Called from** *[output()](output)*
>>
>> **Call to** *[cc2real()](cc2real)*, *[lmas2pt()](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](l_max)*+1,*nrstop*-(*nrstart*)+1) *[real,in]*
>>>
>>> - **eparlmr** (*[l_max](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 `curlu2meanr`, `buomeanr`, `ohmdissr`

Routines `initialize_output_power()`, `get_power()`

Needed modules

- `blocking` (`lmstartb()`, `lmstopb()`, `lo_map()`, `st_map()`): Module containing blocking information

- `num_param` (`escale()`, `tscale()`): Module containing numerical and control parameters

- `horizontal_data` (`dlh()`): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

- `parallel_mod`: This module contains the blocking information

- `radial_data` (`n_r_cmb()`, `n_r_icb()`)

- `output_data` (`power_file()`, `tag()`, `n_power_file()`): This module contains the parameters for output control

- `radial_functions` (`r_ic()`, `or2()`, `drx()`, `dr_fac_ic()`, `o_r_ic()`, `d_costf1_ic_init()`, `r_cmb()`, `i_costf_init()`, `o_r_ic2()`, `r()`, `d_costf_init()`, `r_icb()`, `rgrav()`, `lambda()`, `i_costf1_ic_init()`): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- `logic` (`l_rot_ma()`, `l_save_out()`, `l_mag()`, `l_conv()`, `l_rot_ic()`, `l_heat()`, `l_sric()`, `l_cond_ic()`, `l_srma()`): Module containing the logicals that control the run

- `useful` (`cc2real()`, `cc22real()`): library with several useful subroutines

- `outrot` (`get_viscous_torque()`)

- `truncation` (`n_r_ic_maxmag()`, `n_r_max()`, `n_r_ic_max()`, `n_r_maxmag()`): This module defines the grid points and the truncation

- `precision_mod`: This module controls the precision used in MagIC

- `constants` (`two()`, `one()`, `half()`): module containing constants and parameters used in the code.

- `physical_parameters` (`lffac()`, `ktopv()`, `opm()`, `kbotv()`): Module containing the physical parameters

- `lmloop_data` (`llm()`, `ulmmag()`, `llmmag()`, `ulm()`)

- `integration` (`rint_r()`, `rintic()`): Radial integration functions

Variables

- power/**curlu2meanr** (:) *[real,private/allocatable]*

- power/**ohmdissr** (:) *[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 `ek_t_m_ave`, `e_cmb_l_ave`, `dt_icb2_ave`, `e_cmb2_l_ave`, `e_p_m_ave`, `ek_t2_m_ave`, `t_icb_ave`, `t_ave`, `e_cmb2_m_ave`, `e_t_m_ave`, `e_p_l_ave`, `ek_p2_m_ave`, `e_p2_m_ave`, `ek_t2_l_ave`, `t2_ave`, `dt_icb_ave`, `ek_p2_l_ave`, `e_t2_l_ave`, `e_t2_m_ave`, `e_p2_l_ave`, `ek_t_l_ave`, `ek_p_l_ave`, `ek_p_m_ave`, `e_cmb_m_ave`, `e_t_l_ave`, `t_icb2_ave`

Routines `get_standard_deviation()`, `spectrum_average()`, `spectrum_temp()`, `spectrum_temp_average()`, `spectrum()`, `initialize_spectra()`

Needed modules

- `blocking` (`lo_map()`, `st_map()`): Module containing blocking information

- `num_param` (`tscale()`, `escale()`): Module containing numerical and control parameters

- `horizontal_data` (`dlh()`): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

- `parallel_mod`: This module contains the blocking information

- `radial_data` (`n_r_cmb()`, `n_r_icb()`)

- `output_data` (`tag()`, `log_file()`, `nlf()`, `n_kin_spec_file()`, `n_mag_spec_file()`, `n_u2_spec_file()`): This module contains the parameters for output control

- `radial_functions` (`d_costf1_ic_init()`, `orho2()`, `drx()`, `dr_fac_ic()`, `dr_fac()`, `d_costf_init()`, `r_ic()`, `i_costf_init()`, `r()`, `orho1()`, `r_icb()`, `or2()`, `i_costf1_ic_init()`): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- `logic` (`l_anel()`, `l_cond_ic()`, `l_heat()`, `l_mag()`): Module containing the logicals that control the run

- `useful` (`safeopen()`, `cc2real()`, `safeclose()`, `cc22real()`): library with several useful subroutines

- `truncation` (`n_r_ic_maxmag()`, `n_r_max()`, `l_max()`, `n_r_maxmag()`, `minc()`, `n_r_ic_max()`): This module defines the grid points and the truncation

- `precision_mod`: This module controls the precision used in MagIC

- `constants` (`vol_oc()`, `pi()`, `one()`, `four()`, `half()`): module containing constants and parameters used in the code.

- `physical_parameters` (`lffac()`): Module containing the physical parameters

- *lmloop_data* (*llm()*, *ulmmag()*, *llmmag()*, *ulm()*)
- *integration* (*rint_r()*, *rint()*, *rintic()*): Radial integration functions

Variables

- spectra/**ek_t2_l_ave** (:) *[real,private/allocatable]*
- spectra/**t_icb_ave** (:) *[real,private/allocatable]*
- spectra/**e_cmb2_m_ave** (:) *[real,private/allocatable]*
- spectra/**e_t2_m_ave** (:) *[real,private/allocatable]*
- spectra/**ek_p2_l_ave** (:) *[real,private/allocatable]*
- spectra/**e_cmb2_l_ave** (:) *[real,private/allocatable]*
- spectra/**e_p_l_ave** (:) *[real,private/allocatable]*
- spectra/**e_p2_m_ave** (:) *[real,private/allocatable]*
- spectra/**e_t_l_ave** (:) *[real,private/allocatable]*
- spectra/**e_t2_l_ave** (:) *[real,private/allocatable]*
- spectra/**t_icb2_ave** (:) *[real,private/allocatable]*
- spectra/**e_t_m_ave** (:) *[real,private/allocatable]*
- spectra/**dt_icb2_ave** (:) *[real,private/allocatable]*
- spectra/**e_p2_l_ave** (:) *[real,private/allocatable]*
- spectra/**e_p_m_ave** (:) *[real,private/allocatable]*
- spectra/**dt_icb_ave** (:) *[real,private/allocatable]*
- spectra/**e_cmb_m_ave** (:) *[real,private/allocatable]*
- spectra/**e_cmb_l_ave** (:) *[real,private/allocatable]*
- spectra/**ek_p2_m_ave** (:) *[real,private/allocatable]*
- spectra/**ek_p_l_ave** (:) *[real,private/allocatable]*
- spectra/**t2_ave** (:) *[real,private/allocatable]*
- spectra/**ek_p_m_ave** (:) *[real,private/allocatable]*
- spectra/**t_ave** (:) *[real,private/allocatable]*
- spectra/**ek_t2_m_ave** (:) *[real,private/allocatable]*
- spectra/**ek_t_l_ave** (:) *[real,private/allocatable]*
- spectra/**ek_t_m_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 *output()*, *fields_average()*

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** *output()*, *fields_average()*
>
> **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()*, *graphout_mpi_header()*, *graph_write()*, *graph_write_mpi()*, *graphout_mpi()*, *graphout()*

Needed modules

- *blocking* (*sizethetab()*, *nfs()*, *nthetabs()*): Module containing blocking information
- *num_param* (*vscale()*): Module containing numerical and control parameters
- *horizontal_data* (*plm()*, *dplm()*, *dlh()*, *theta_ord()*, *o_sin_theta()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *parallel_mod*: This module contains the blocking information

- *radial_data* (*n_r_icb()*)

- *output_data* (*runid()*, *graph_mpi_fh()*, *n_graph_file()*): This module contains the parameters for output control

- *radial_functions* (*r_ic()*, *r_cmb()*, *or2()*, *o_r_ic2()*, *r()*, *orho1()*, *r_icb()*, *o_r_ic()*, *or1()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *leg_helper_mod* (*legprep_ic()*)

- *logic* (*l_cond_ic()*, *l_mag()*): Module containing the logicals that control the run

- *legendre_spec_to_grid* (*legtf()*)

- *truncation* (*n_r_ic_maxmag()*, *n_r_maxmag()*, *l_max()*, *lm_maxmag()*, *n_theta_max()*, *n_r_max()*, *n_phi_tot()*, *nrp()*, *n_phi_max()*, *lm_max()*, *minc()*, *n_r_ic_max()*): This module defines the grid points and the truncation

- *precision_mod*: This module controls the precision used in MagIC

- *physical_parameters* (*pr()*, *sigma_ratio()*, *radratio()*, *ek()*, *ra()*, *prmag()*): Module containing the physical parameters

- *fft*: This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5

Variables

Subroutines and functions

subroutine graphout_mod/**graphout** (*time*, *n_r*, *vr*, *vt*, *vp*, *br*, *bt*, *bp*, *sr*, *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]*
> >
> > - **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_mpi**(*time*, *n_r*, *vr*, *vt*, *vp*, *br*, *bt*, *bp*, *sr*, *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]*
- **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_seq(), *do_iteration_thetablocking_openmp()*

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 *output()*, *fields_average()*

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_ic()*, *graphout_mpi()*

10.16.2 `movie.f90`

Quick access

Variables *n_movie_fields_max*, *n_movies_max*, *movie*, *movie_file*, *movie_const*, *frames*, *t_movies*, *n_movie_fields*, *n_movie_const*, *n_movie_type*, *n_movie_fields_ic*, *lstoremov*, *n_movie_file*, *licfield*, *n_movie_field_type*, *n_movie_field_start*, *n_movie_field_stop*, *n_movie_surface*, *moviedipcolat*, *moviedipstrengthgeo*, *moviedipstrength*, *moviediplon*, *n_frame_work*, *n_md*, *n_movies*

Routines *movie_gather_frames_to_rank0()*, *finalize_movie_data()*, *initialize_movie_data()*, *get_movie_type()*

Needed modules

- *output_data* (*log_file()*, *tag()*, *n_log_file()*): This module contains the parameters for output control

- *constants* (*pi()*, *one()*): module containing constants and parameters used in the code.

- *precision_mod*: This module controls the precision used in MagIC

- *horizontal_data* (*theta()*, *phi()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

- *parallel_mod*: This module contains the blocking information

- *radial_data* (*nrstart()*, *n_r_cmb()*, *n_r_icb()*, *nrstop()*)

- *truncation* (*n_theta_max()*, *n_r_max()*, *ldtbmem()*, *n_phi_max()*, *lmoviemem()*, *minc()*, *n_r_ic_max()*, *n_r_tot()*): This module defines the grid points and the truncation

- *radial_functions* (*r_ic()*, *r_cmb()*, *r()*, *r_icb()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *charmanip* (*delete_string()*, *capitalize()*, *length_to_blank()*, *str2dble()*): This module contains several useful routines to manipule character strings

- *logic* (*l_htmovie()*, *l_save_out()*, *l_store_frame()*, *l_movie()*, *l_movie_oc()*, *l_movie_ic()*, *l_dtbmovie()*): Module containing the logicals that control the run

- *useful* (*logwrite()*): library with several useful subroutines

Variables

- movie_data/**movie** (30) *[character,public]*
    Only for input

- movie_data/**n_movies_max** *[integer,parameter=30/public]*
    Max no. of different movies

- movie_data/**licfield** (30) *[logical,public]*

- movie_data/**moviedipstrengthgeo** *[real,public]*

- movie_data/**n_frame_work** *[integer,public]*

- movie_data/**n_movie_field_type** (6,30) *[integer,public]*

- movie_data/**movie_const** (30) *[real,public]*

- movie_data/**n_movie_field_start** (6,30) *[integer,public]*

- movie_data/**n_movie_fields_ic** (30) *[integer,public]*

- movie_data/**n_movie_const** (30) *[integer,public]*

- movie_data/**n_movie_surface** (30) *[integer,public]*

- movie_data/**n_movie_type** (30) *[integer,public]*

- movie_data/**movie_file** (30) *[character,public]*

- movie_data/**moviediplon** *[real,public]*

- movie_data/**moviedipstrength** *[real,public]*

- movie_data/**n_movies** *[integer,public]*

- movie_data/**n_movie_file** (30) *[integer,public]*

- movie_data/**n_md** *[integer,public]*

- movie_data/**t_movies** (10000) *[real,public]*

- movie_data/**n_movie_fields_max** *[integer,parameter=6/public]*
  Max no. of fields per movie

- movie_data/**n_movie_field_stop** (6,30) *[integer,public]*

- movie_data/**frames** (:) *[real,allocatable/public]*

- movie_data/**moviedipcolat** *[real,public]*

- movie_data/**n_movie_fields** (30) *[integer,public]*

- movie_data/**lstoremov** (30) *[logical,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

10.16. IO: graphic files, movie files, coeff files and potential files   **297**

- •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** *store_fields_3d()*, *get_fl()*, *store_movie_frame()*, *store_fields_p()*,
> *store_fields_t()*, *get_sl()*, *get_b_surface()*, *store_fields_r()*,
> *store_fields_sur()*, *write_movie_frame()*

Needed modules

- *blocking* (*nfs()*, *lm2l()*, *lm2()*): Module containing blocking information

- *num_param* (*tscale()*, *vscale()*): Module containing numerical and control parameters

- *horizontal_data* (*osn1()*, *phi()*, *theta_ord()*, *n_theta_cal2ord()*, *dlh()*, *dplm()*,
  costheta(), *sintheta()*, *o_sin_theta()*, *plm()*, *dphi()*, *d_l()*, *o_sin_theta_e2()*):
  Module containing functions depending on longitude and latitude plus help arrays depending on degree and
  order

- *fields* (*b()*, *w_rloc()*, *b_ic()*, *b_rloc()*): This module contains the potential fields and their radial
  derivatives

- *radial_data* (*n_r_icb()*)

- *truncation* (*n_theta_max()*, *n_r_maxmag()*, *l_max()*, *lm_maxmag()*, *n_m_max()*, *nrp()*,
  n_r_ic_maxmag(), *n_phi_max()*, *lm_max()*, *minc()*, *n_r_ic_max()*, *n_r_max()*): This mod-
  ule defines the grid points and the truncation

- *radial_functions* (*r_ic()*, *r_surface()*, *or4()*, *orho2()*, *or1()*, *r()*, *r_cmb()*, *orho1()*,
  or3(), *or2()*, *beta()*): This module initiates all the radial functions (transport properties, density, temper-
  ature, cheb transforms, etc.)

- *out_dtb_frame* (*write_dtb_frame()*)

- *logic* (*l_cond_ic()*, *l_save_out()*): Module containing the logicals that control the run

- *output_data* (*runid()*): This module contains the parameters for output control

- *precision_mod*: This module controls the precision used in MagIC

- *physical_parameters* (*pr()*, *lffac()*, *radratio()*, *ek()*, *ra()*, *prmag()*): Module containing
  the physical parameters

- *movie_data* (*n_movie_file()*, *moviedipstrength()*, *frames()*, *movie_file()*,
  moviedipstrengthgeo(), *lstoremov()*, *n_movie_type()*, *n_movie_fields()*,
  t_movies(), *movie_const()*, *n_movie_field_start()*, *moviedipcolat()*,
  n_movie_const(), *n_movie_surface()*, *moviediplon()*, *n_movie_field_type()*,
  n_movie_fields_ic(), *n_movies()*, *n_movie_field_stop()*)

- *fft* (*fft_thetab()*): This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3,
  wpass4 and wpass5

- *constants* (*two()*, *one()*, *zero()*): module containing constants and parameters used in the code.

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_seq(), *do_iteration_thetablocking_openmp()*
>
> **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_movie_frame_ic()*, *store_fields_p()*

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

- *blocking* (*sizethetab()*, *nfs()*, *nthetabs()*): Module containing blocking information
- *horizontal_data* (*dplm()*, *n_theta_cal2ord()*, *dlh()*, *plm()*, *o_sin_theta()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *radial_data* (*n_r_icb()*)
- *truncation* (*n_r_ic_maxmag()*, *n_phi_max()*, *l_max()*, *n_r_ic_max()*, *lm_maxmag()*, *n_theta_max()*, *n_r_maxmag()*, *nrp()*, *minc()*, *lm_max()*): This module defines the grid points and the truncation
- *radial_functions* (*r_ic()*, *r_icb()*, *o_r_ic()*, *o_r_ic2()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *leg_helper_mod* (*legprep_ic()*)
- *logic* (*l_cond_ic()*): Module containing the logicals that control the run
- *legendre_spec_to_grid* (*legtf()*)
- *out_movie* (*get_fl()*)
- *precision_mod*: This module controls the precision used in MagIC
- *physical_parameters* (*lffac()*): Module containing the physical parameters

- *movie_data* (*n_movie_field_stop()*, *n_movie_field_start()*, *n_movie_fields()*, *n_movie_fields_ic()*, *n_movie_const()*, *n_movie_surface()*, *n_movie_type()*, *n_movie_field_type()*, *n_movies()*, *frames()*)

- *fft* (*fft_thetab()*): This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5

- *constants* (*one()*): module containing constants and parameters used in the code.

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_coeff_r()*, *write_bcmb()*

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** *output()*, *fields_average()*

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 *storepotw()*, *storepot()*

Needed modules

- *output_data* (*tag()*): This module contains the parameters for output control
- *communications* (*gather_from_lo_to_rank0()*)
- *precision_mod*: This module controls the precision used in MagIC
- *cosine_transform* (*costf1()*)
- *parallel_mod* (rank()): This module contains the blocking information
- *physical_parameters* (*pr()*, *sigma_ratio()*, *radratio()*, *ek()*, *ra()*, *prmag()*): Module containing the physical parameters
- *lmloop_data* (*llm()*, *ulm()*)
- *truncation* (*n_r_max()*, *l_max()*, *n_cheb_max()*, *lm_max()*, *minc()*, *n_r_ic_max()*, *n_cheb_ic_max()*): This module defines the grid points and the truncation
- *radial_functions* (*d_costf1_ic_init()*, *i_costf_init()*, *i_costf1_ic_init()*, *d_costf_init()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *logic* (*l_cond_ic()*): Module containing the logicals that control the run
- *constants* (*two()*, *half()*): module containing constants and parameters used in the code.

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*, *worka*, *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]*

- **worka** (*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** `b_ic_ave`, `s_ave_global`, `z_ave_global`, `b_ave`, `w_ave_global`, `dw_ave_global`, `aj_ave`, `w_ave`, `s_ave`, `z_ave`, `aj_ic_ave`, `aj_ave_global`, `db_ave_global`
>
> **Routines** `initialize_fields_average_mod()`, `fields_average()`

Needed modules

- `blocking` (`nfs()`, `lmstartb()`, `sizethetab()`, `nthetabs()`, `lmstopb()`, `lm2()`): Module containing blocking information

- `communications` (`gt_ic()`, `gt_oc()`, `gather_all_from_lo_to_rank0()`, `gather_from_lo_to_rank0()`, `get_global_sum()`)

- `spectra` (`spectrum()`, `spectrum_temp()`)

- `graphout_mod` (`graphout()`, `graphout_ic()`)

- `out_coeff` (`write_bcmb()`): This module contains the subroutines that calculate the Bcmb files and the [B|V|T]_coeff_r files

- `truncation`: This module defines the grid points and the truncation

- `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)

- `kinetic_energy` (`get_e_kin()`)

- `radial_der` (`get_drns()`): Radial derivatives functions

- `fft`: This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5

- *magnetic_energy* (*get_e_mag()*)

- *radial_data* (*n_r_cmb()*)

- *output_data* (*tag()*, *log_file()*, *l_max_cmb()*, *n_graph_file()*, *nlf()*, *n_graphs()*, *graph_file()*): This module contains the parameters for output control

- *radial_functions* (*d_costf1_ic_init()*, *i_costf_init()*, *drx()*, *d_costf2_ic_init()*, *r()*, *d_costf_init()*, *i_costf2_ic_init()*, *i_costf1_ic_init()*, *dr_fac_ic()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *leg_helper_mod* (*legprep()*)

- *logic* (*l_save_out()*, *l_heat()*, *l_mag()*, *l_cond_ic()*, *l_conv()*): Module containing the logicals that control the run

- *legendre_spec_to_grid* (*legtf()*)

- *radial_der_even* (*get_drns_even()*, *get_ddrns_even()*)

- *precision_mod*: This module controls the precision used in MagIC

- *horizontal_data* (*dplm()*, *dlh()*, *plm()*): 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

- *lmloop_data* (*llm()*, *ulmmag()*, *llmmag()*, *ulm()*)

- *constants* (*vol_ic()*, *vol_oc()*, *one()*, *zero()*): module containing constants and parameters used in the code.

Variables

- fields_average_mod/**s_ave** (:,:) *[complex,private/allocatable]*

- fields_average_mod/**b_ic_ave** (:,:) *[complex,private/allocatable]*

- fields_average_mod/**b_ave** (:,:) *[complex,private/allocatable]*

- fields_average_mod/**dw_ave_global** (:) *[complex,private/allocatable]*

- fields_average_mod/**z_ave_global** (:) *[complex,private/allocatable]*

- fields_average_mod/**aj_ave** (:,:) *[complex,private/allocatable]*

- fields_average_mod/**aj_ave_global** (:) *[complex,private/allocatable]*

- fields_average_mod/**s_ave_global** (:) *[complex,private/allocatable]*

- fields_average_mod/**z_ave** (:,:) *[complex,private/allocatable]*

- fields_average_mod/**db_ave_global** (:) *[complex,private/allocatable]*

- fields_average_mod/**w_ave** (:,:) *[complex,private/allocatable]*

- fields_average_mod/**aj_ic_ave** (:,:) *[complex,private/allocatable]*

- fields_average_mod/**w_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*, *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]*
> - **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 global array used when RMS force balance is requested

Quick access

> **Variables** *lfpolas2hint*, *dtvpol2hint*, *advpollmr*, *difpollmr*, *lfpol2hint*, *cortor2hint*, *cortoras2hint*, *lftor2hint*, *advpol2hint*, *geo2hint*,

---

> > *geoas2hint*, *arcas2hint*, *dtbpol2hint*, *mag2hint*, *corpol2hint*, *buolmr*, *dtbpollmr*, *dtvpollmr*, *dtbtor2hint*, *corpolas2hint*, *dtbpolas2hint*, *maglmr*, *dtvtoras2hint*, *lftoras2hint*, *diftor2hint*, *arclmr*, *dtvpolas2hint*, *diftoras2hint*, *advtoras2hint*, *dtvtor2hint*, *magas2hint*, *buoas2hint*, *lfpollmr*, *corpollmr*, *dtbtoras2hint*, *difpolas2hint*, *buo2hint*, *prelmr*, *geolmr*, *arc2hint*, *advpolas2hint*, *preas2hint*, *difpol2hint*, *advtor2hint*, *pre2hint*

> **Routines** *zerorms()*, *initialize_rms()*

Needed modules

- *truncation* (*n_r_max()*, *lm_max()*, *lm_maxmag()*, *n_r_maxmag()*): This module defines the grid points and the truncation
- *precision_mod*: This module controls the precision used in MagIC
- *constants* (*zero()*): module containing constants and parameters used in the code.

Variables

- rms/**lfpol2hint** (:) *[real,allocatable/public]*
- rms/**dtvpollmr** (:,:) *[complex,allocatable/public]*
- rms/**dtbtor2hint** (:,:) *[real,allocatable/public]*
- rms/**difpol2hint** (:,:) *[real,allocatable/public]*
- rms/**difpollmr** (:,:) *[complex,allocatable/public]*
- rms/**buolmr** (:,:) *[complex,allocatable/public]*
- rms/**dtbtoras2hint** (:,:) *[real,allocatable/public]*
- rms/**difpolas2hint** (:,:) *[real,allocatable/public]*
- rms/**dtvtor2hint** (:,:) *[real,allocatable/public]*
- rms/**arc2hint** (:) *[real,allocatable/public]*
- rms/**buo2hint** (:) *[real,allocatable/public]*
- rms/**geoas2hint** (:) *[real,allocatable/public]*
- rms/**advtor2hint** (:) *[real,allocatable/public]*
- rms/**buoas2hint** (:) *[real,allocatable/public]*
- rms/**advpolas2hint** (:) *[real,allocatable/public]*
- rms/**pre2hint** (:) *[real,allocatable/public]*
- rms/**dtbpolas2hint** (:,:) *[real,allocatable/public]*
- rms/**arcas2hint** (:) *[real,allocatable/public]*
- rms/**lfpollmr** (:,:) *[complex,allocatable/public]*
- rms/**advtoras2hint** (:) *[real,allocatable/public]*
- rms/**dtvpol2hint** (:,:) *[real,allocatable/public]*
- rms/**arclmr** (:,:) *[complex,allocatable/public]*

- rms/**lfpolas2hint** (:) *[real,allocatable/public]*
- rms/**maglmr** (:,:) *[complex,allocatable/public]*
- rms/**cortor2hint** (:) *[real,allocatable/public]*
- rms/**cortoras2hint** (:) *[real,allocatable/public]*
- rms/**magas2hint** (:) *[real,allocatable/public]*
- rms/**lftor2hint** (:) *[real,allocatable/public]*
- rms/**advpol2hint** (:) *[real,allocatable/public]*
- rms/**prelmr** (:,:) *[complex,allocatable/public]*
- rms/**dtbpollmr** (:,:) *[complex,allocatable/public]*
- rms/**dtvtoras2hint** (:,:) *[real,allocatable/public]*
- rms/**advpollmr** (:,:) *[complex,allocatable/public]*
- rms/**dtvpolas2hint** (:,:) *[real,allocatable/public]*
- rms/**dtbpol2hint** (:,:) *[real,allocatable/public]*
- rms/**diftoras2hint** (:,:) *[real,allocatable/public]*
- rms/**corpollmr** (:,:) *[complex,allocatable/public]*
- rms/**lftoras2hint** (:) *[real,allocatable/public]*
- rms/**mag2hint** (:) *[real,allocatable/public]*
- rms/**diftor2hint** (:,:) *[real,allocatable/public]*
- rms/**geolmr** (:,:) *[complex,allocatable/public]*
- rms/**corpolas2hint** (:) *[real,allocatable/public]*
- rms/**preas2hint** (:) *[real,allocatable/public]*
- rms/**corpol2hint** (:) *[real,allocatable/public]*
- rms/**geo2hint** (:) *[real,allocatable/public]*

Subroutines and functions

subroutine rms/**initialize_rms**()

> Memory allocation

> > **Called from** *magic*

subroutine rms/**zerorms**()

> Zeros integrals that are set in get_td, update_z, update_wp, update_b, dtVrms and dtBrms

> > **Called from** *output()*

10.17.2 `RMS_helpers.f90`

Description

This module contains several useful subroutines required to compute RMS diagnostics

Quick access

> **Routines** *hint2tor()*, *hint2dpol()*, *get_paslm()*, *hint2pol()*, *get_ras()*, *get_poltorrms()*, *init_rnb()*

Needed modules

- *blocking* (*st_map()*, *lm2()*): Module containing blocking information
- *horizontal_data* (*dplm()*, *dlh()*, *plm()*, *osn1()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *truncation* (*lm_max_dtb()*, *n_r_max()*, *l_max()*, *lm_max()*): This module defines the grid points and the truncation
- *radial_functions* (*d_costf_init()*, *or2()*, *drx()*, *r()*, *i_costf_init()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *chebyshev_polynoms_mod* (*cheb_grid()*)
- *radial_der* (*get_dr()*): Radial derivatives functions
- *precision_mod*: This module controls the precision used in MagIC
- *integration* (*rint_r()*): Radial integration functions
- *init_costf* (*init_costf1()*)
- *lmmapping* (mappings())
- *useful* (*cc2real()*): library with several useful subroutines
- *constants* (*vol_oc()*, *one()*): module containing constants and parameters used in the code.

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*, *polas2hint*, *map*)

Parameters

- **dpol** (*lm_max*) *[complex,in]* :: Toroidal field Potential
- **lmstart** *[integer,in]*
- **lmstop** *[integer,in]*
- **pol2hint** *[real,inout]*
- **polas2hint** *[real,inout]*
- **map** *[mappings,in]*

Called from *dtvrms()*, *dtbrms()*

Call to *cc2real()*

subroutine rms_helpers/**hint2pol**(*pol*, *lb*, *ub*, *nr*, *lmstart*, *lmstop*, *pollmr*, *pol2hint*, *polas2hint*, *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*,*n_r_max*) *[complex,out]*
- **pol2hint** *[real,inout]*
- **polas2hint** *[real,inout]*

- **map** *[mappings,in]*

Called from *get_td()*, *updatewp()*, *updateb()*

Call to *cc2real()*

subroutine rms_helpers/**hint2tor**(*tor*, *lb*, *ub*, *nr*, *lmstart*, *lmstop*, *tor2hint*, *toras2hint*, *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** *[real,inout]*

- **toras2hint** *[real,inout]*

- **map** *[mappings,in]*

Called from *updatez()*, *get_td()*, *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()*

subroutine rms_helpers/**init_rnb**(*r*, *n_r_max*, *n_cheb_max*, *rcut*, *rdea*, *r2*, *n_r_max2*, *n_cheb_max2*, *ns*, *dr_fac2*, *i_costf_init2*, *ndi_costf1*, *d_costf_init2*, *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** (*) *[real,in]*

- **n_r_max** *[integer,in]*

- **n_cheb_max** *[integer,in]*

- **rcut** *[real,in]*
- **rdea** *[real,in]*
- **r2** (*) *[real,out]*
- **n_r_max2** *[integer,out]*
- **n_cheb_max2** *[integer,out]*
- **ns** *[integer,out]*
- **dr_fac2** (*) *[real,out]*
- **i_costf_init2** (*ndi_costf1*) *[integer,out]* :: info for transform
- **ndi_costf1** *[integer,in]*
- **d_costf_init2** (*ndd_costf1*) *[real,out]* :: info for tranfor
- **ndd_costf1** *[integer,in]*

Called from *dtvrms()*

Call to *cheb_grid()*, *init_costf1()*

10.17.3 `out_RMS.f90`

Quick access

Routines *dtvrms()*, *dtbrms()*

Needed modules

- *blocking* (*sizethetab()*, *nfs()*, *lm2()*, *nthetabs()*, *st_map()*, *lo_map()*): Module containing blocking information
- *dtb_mod* (*tadvrms()*, *pstrasrms()*, *tdifasrms()*, *tdiflm()*, *tstrasrms()*, *padvlm()*, *tomelm()*, *tadvasrms()*, *pstrlm()*, *tomeasrms()*, *tstrlm()*, *padvasrms()*, *pdifrms()*, *pstrrms()*, *padvrms()*, *tdifrms()*, *pdifasrms()*, *tomerms()*, *pdiflm()*, *tadvlm()*, *tstrrms()*): This module contains magnetic field stretching and advection terms plus a separate omega-effect. It is used for movie output....
- *num_param* (*tscale()*): Module containing numerical and control parameters
- *horizontal_data* (*theta_ord()*, *phi()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *parallel_mod*: This module contains the blocking information
- *radial_data* (*nrstart()*, *nrstop()*)
- *communications* (*myallgather()*)
- *truncation* (*lm_max_dtb()*, *lm_max()*, *minc()*, *lm_maxmag()*, *n_theta_max()*, *n_r_max()*, *n_r_max_dtb()*, *n_cheb_max()*, *n_r_maxmag()*, *n_phi_max()*): This module defines the grid points and the truncation
- *radial_functions* (*drx()*, *rc()*, *i_costf_initc()*, *n_r_maxc()*, *n_cheb_maxc()*, *r_cmb()*, *i_costf_init()*, *rgrav()*, *dr_facc()*, *r()*, *d_costf_init()*, *ncut()*, *ndd_costf1()*, *d_costf_initc()*, *ndi_costf1()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *logic* (*l_mag_lf()*, *l_save_out()*, *l_rmstest()*, *l_conv()*, *l_conv_nl()*, *l_heat()*, *l_corr()*): Module containing the logicals that control the run

- *output_data* (*rcut()*, *dtbrms_file()*, *n_dtbrms_file()*, *runid()*, *dtvrms_file()*, *rdea()*, *dtvasrms_file()*, *tag()*, *n_dtdrms_file()*, *dtdrms_file()*, *n_dtvrms_file()*, *n_dtvasrms_file()*): This module contains the parameters for output control

- *radial_der* (*get_drns()*): Radial derivatives functions

- *precision_mod*: This module controls the precision used in MagIC

- *integration* (*rint_r()*): Radial integration functions

- *physical_parameters* (*ek()*, *pr()*, *ra()*, *prmag()*, *radratio()*): Module containing the physical parameters

- *rms_helpers* (*get_poltorrms()*, *hint2dpol()*, *get_paslm()*, *init_rnb()*, *get_ras()*): This module contains several useful subroutines required to compute RMS diagnostics

- *constants* (*third()*, *four()*, *vol_oc()*, *zero()*, *pi()*, *half()*): module containing constants and parameters used in the code.

- *rms* (*lfpol2hint()*, *cortor2hint()*, *geoas2hint()*, *difpol2hint()*, *lftoras2hint()*, *buolmr()*, *geolmr()*, *difpolas2hint()*, *dtvtor2hint()*, *arc2hint()*, *buo2hint()*, *dtbtor2hint()*, *advtor2hint()*, *buoas2hint()*, *advpolas2hint()*, *pre2hint()*, *dtbpolas2hint()*, *arcas2hint()*, *lfpollmr()*, *advtoras2hint()*, *dtvpol2hint()*, *arclmr()*, *lfpolas2hint()*, *maglmr()*, *cortoras2hint()*, *magas2hint()*, *lftor2hint()*, *advpol2hint()*, *prelmr()*, *dtbpollmr()*, *dtvpollmr()*, *dtvtoras2hint()*, *advpollmr()*, *dtvpolas2hint()*, *dtbpol2hint()*, *diftoras2hint()*, *corpollmr()*, *difpollmr()*, *mag2hint()*, *diftor2hint()*, *dtbtoras2hint()*, *corpolas2hint()*, *preas2hint()*, *corpol2hint()*, *geo2hint()*): This module contains the global array used when RMS force balance is requested

Variables

Subroutines and functions

subroutine out_rms/**dtvrms** (*time*, *nrms_sets*)

> For testing RMS balance and determining the necessary values of rDea and rCut one can use the l_RMStest=true options. In this case the poloidal and toroidal RMS dtV which also include the diffusion effects should be identical (as close as desired) to the RMS sum of forces stored in the Geo value and the Mag value, respectively. An additional tests is the Arc value which should be identical to the poloidal kinetic energy.

> **Note:** The second test with the Arc value cannot work so easily in the anelastic version as the density enters now in the force balance. The first test should work, though.

> **Parameters**
> - **time** *[real,in]*
> - **nrms_sets** *[integer,inout]*

> **Called from** *output()*

> **Call to** *myallgather()*, *init_rnb()*, *get_drns()*, *hint2dpol()*, *rint_r()*

subroutine out_rms/**dtbrms** (*time*)

Parameters time *[real,in]*

Called from *output()*

Call to *myallgather()*, *get_drns()*, *get_poltorrms()*, *hint2dpol()*, *rint_r()*, *get_ras()*, *get_paslm()*

10.17.4 `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 *tstrrlm_rloc*, *pdiflmic*, *pdiflm_lmloc*, *tadvlmic_lmloc*, *tadvlmic*, *tadvrlm_rloc*, *tomerlm_rloc*, *tomerlm*, *pstrlm_rloc*, *padvlm*, *pdiflmic_lmloc*, *tdiflm_lmloc*, *tadvlm*, *tstrlm_rloc*, *pstrlm*, *tstrlm*, *padvlmic_lmloc*, *tadvlm_rloc*, *tstrrlm*, *padvlm_rloc*, *pdiflm*, *tomelm*, *tdiflmic*, *tadvrlm*, *tomelm_rloc*, *tdiflm*, *padvlmic*, *tdiflmic_lmloc*, *tstrasrms*, *tomerms*, *tadvrms*, *pstrrms*, *pdifrms*, *tstrrms*, *padvrms*, *pstrasrms*, *tdifrms*, *tadvasrms*, *pdifasrms*, *tomeasrms*, *tdifasrms*, *padvasrms*

Routines *dtb_gather_rloc_on_rank0()*, *get_dtblm()*, *get_dh_dtblm()*, *initialize_dtb_mod()*, *get_dtblmfinish()*

Needed modules

- *blocking* (*lm2l()*, *lmp2lmpa()*, *l2lmas()*, *lm2m()*, *nfs()*, *lm2lmp()*, *lmp2lmps()*, *lo_map()*, *st_map()*): Module containing blocking information

- *communications* (*gt_ic()*, *gt_oc()*, *gather_all_from_lo_to_rank0()*)

- *horizontal_data* (*dtheta1s()*, *hdif_b()*, *dphi()*, *osn1()*, *osn2()*, *d_lp1()*, *cosn2()*, *dlh()*, *dtheta1a()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

- *physical_parameters* (*o_sr()*, *opm()*): Module containing the physical parameters

- *radial_data* (*nrstart()*, *nrstop()*)

- *truncation* (*n_r_ic_maxmag()*, *n_r_maxmag()*, *l_max()*, *n_phi_max()*, *n_r_ic_max_dtb()*, *n_r_max()*, *n_r_max_dtb()*, *n_cheb_max()*, *ldtbmem()*, *lm_max_dtb()*, *lm_max()*, *nrp()*, *n_r_ic_max()*): This module defines the grid points and the truncation

- *radial_functions* (*or2()*, *drx()*, *i_costf_init()*, *orho1()*, *o_r_ic()*, *lambda()*, *or1()*, *dllambda()*, *d_costf_init()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *logic* (*l_cond_ic()*, *l_dtrmagspec()*): Module containing the logicals that control the run

- *radial_der* (*get_drns()*): Radial derivatives functions

- *precision_mod*: This module controls the precision used in MagIC

- *parallel_mod*: This module contains the blocking information

- *radial_spectra*

- *lmloop_data* (*ulmmag()*, *llm()*, *llmmag()*, *ulm()*)

- *legendre_grid_to_spec* (*legtf3()*, *legtf2()*)

- *fft*: This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5

- *constants* (*two()*): module containing constants and parameters used in the code.

Variables

- dtb_mod/**tadvrms** *[real,public]*

- dtb_mod/**pdiflmic** (:,:) *[complex,allocatable/public]*

- dtb_mod/**tdiflm** (:,:) *[complex,allocatable/public]*

- dtb_mod/**pstrlm** (:,:) *[complex,allocatable/public]*

- dtb_mod/**tomeasrms** *[real,public]*

- dtb_mod/**tadvasrms** *[real,public]*

- dtb_mod/**padvlmic_lmloc** (:,:) *[complex,allocatable/public]*

- dtb_mod/**pstrrms** *[real,public]*

- dtb_mod/**tdiflm_lmloc** (:,:) *[complex,allocatable/public]*

- dtb_mod/**padvrms** *[real,public]*

- dtb_mod/**tadvlm_rloc** (:,:) *[complex,allocatable/public]*

- dtb_mod/**tdiflmic_lmloc** (:,:) *[complex,allocatable/public]*

- dtb_mod/**pdifasrms** *[real,public]*

- dtb_mod/**pstrasrms** *[real,public]*

- dtb_mod/**tdifasrms** *[real,public]*

- dtb_mod/**pdifrms** *[real,public]*

- dtb_mod/**tomelm** (:,:) *[complex,allocatable/public]*

- dtb_mod/**pdiflmic_lmloc** (:,:) *[complex,allocatable/public]*

- dtb_mod/**tdiflmic** (:,:) *[complex,allocatable/public]*

- dtb_mod/**tadvrlm_rloc** (:,:) *[complex,allocatable/public]*

- dtb_mod/**tomerlm_rloc** (:,:) *[complex,allocatable/public]*

- dtb_mod/**tadvlmic_lmloc** (:,:) *[complex,allocatable/public]*

- dtb_mod/**tstrasrms** *[real,public]*

- dtb_mod/**tomerlm** (:,:) *[complex,allocatable/public]*

- dtb_mod/**tadvlm** (:,:) *[complex,allocatable/public]*

- dtb_mod/**tadvlmic** (:,:) *[complex,allocatable/public]*

- dtb_mod/**padvlmic** (:,:) *[complex,allocatable/public]*

- dtb_mod/**padvasrms** *[real,public]*

- dtb_mod/**pdiflm_lmloc** (:,:) *[complex,allocatable/public]*

- dtb_mod/**tstrrlm** (:,:) *[complex,allocatable/public]*

- dtb_mod/**tstrrms** *[real,public]*

- dtb_mod/**pstrlm_rloc** (:,:) *[complex,allocatable/public]*

- dtb_mod/**padvlm_rloc** (:,:) *[complex,allocatable/public]*

- dtb_mod/**tomelm_rloc** (:,:) *[complex,allocatable/public]*

- dtb_mod/**tstrlm** (:,:) *[complex,allocatable/public]*

- dtb_mod/**tomerms** *[real,public]*

- dtb_mod/**tstrrlm_rloc** (:,:) *[complex,allocatable/public]*

- dtb_mod/**padvlm** (:,:) *[complex,allocatable/public]*

- dtb_mod/**tadvrlm** (:,:) *[complex,allocatable/public]*

- dtb_mod/**tdifrms** *[real,public]*

- dtb_mod/**tstrlm_rloc** (:,:) *[complex,allocatable/public]*

- dtb_mod/**pdiflm** (:,:) *[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](nrp)*,*[nfs](nfs)*) *[real,in]*

               - **vt** (*[nrp](nrp)*,*[nfs](nfs)*) *[real,in]*

               - **vp** (*[nrp](nrp)*,*[nfs](nfs)*) *[real,in]*

               - **br** (*[nrp](nrp)*,*[nfs](nfs)*) *[real,in]*

               - **bt** (*[nrp](nrp)*,*[nfs](nfs)*) *[real,in]*

               - **bp** (*[nrp](nrp)*,*[nfs](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_seq(), *do_iteration_thetablocking_openmp()*

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*, *btvzcotlm*, *btvpsn2lm*, *bpvtsn2lm*, *btvzsn2lm*)

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]*
- **btvzcotlm** (*) *[complex,in]*
- **btvpsn2lm** (*) *[complex,in]*
- **bpvtsn2lm** (*) *[complex,in]*
- **btvzsn2lm** (*) *[complex,in]*

Called from do_iteration_thetablocking_seq(), *do_iteration_thetablocking_openmp()*

10.17.5 `out_dtB_frame.f90`

Quick access

Routines *get_btor()*, *get_dtb()*, *lm2pt()*, *write_dtb_frame()*, *get_bpol()*

Needed modules

- *blocking* (*lm2l()*, *nfs()*, *lm2m()*, *sizethetab()*, *nthetabs()*, *lm2()*): Module containing blocking information
- *dtb_mod* (*tadvlmic()*, *tstrlm()*, *tomelm()*, *tdiflm()*, *pdiflmic()*, *tdiflmic()*, *padvlm()*, *tadvlm()*, *pdiflm()*, *padvlmic()*, *pstrlm()*): This module contains magnetic field stretching and advection terms plus a separate omega-effect. It is used for movie output....
- *horizontal_data* (*plm()*, *osn1()*, *d_lp1()*, *dphi()*, *n_theta_cal2ord()*, *dlh()*, *dplm()*, *costheta()*, *sintheta()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *truncation*: This module defines the grid points and the truncation
- *radial_functions* (*r_ic()*, *drx()*, *d_costf2_ic_init()*, *i_costf2_ic_init()*, *or1()*, *d_costf1_ic_init()*, *i_costf_init()*, *r()*, *d_costf_init()*, *r_icb()*, *i_costf1_ic_init()*, *dr_fac_ic()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *logic* (*l_cond_ic()*): Module containing the logicals that control the run
- *radial_der_even* (*get_drns_even()*)

- *radial_der* (*get_drns()*): Radial derivatives functions

- *precision_mod*: This module controls the precision used in MagIC

- *movie_data* (*n_movie_file()*, *movie_const()*, *n_movie_fields_ic()*, *n_movie_const()*, *n_movie_surface()*, *n_movie_type()*, *n_movie_fields()*, *n_movie_field_type()*)

- *fft*: This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5

- *constants* (*ci()*, *one()*, *zero()*): module containing constants and parameters used in the code.

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 *bspas*, *bpzdas*, *dzcorlmr_rloc*, *dzstrlmr*, *dzddvplmr_rloc*, *bpzas*, *bzpdas*, *dzastrlmr_rloc*, *dzastrlmr*, *bs2as_rloc*, *bpzdas_rloc*, *dzcorlmr*, *dzstrlmr_rloc*, *bspas_rloc*, *v2as*, *dzlflmr_rloc*, *bspdas*, *bspdas_rloc*, *bszas*, *dzlflmr*, *bpzas_rloc*, *bszas_rloc*, *v2as_rloc*, *dzrstrlmr*, *bs2as*, *dzdvplmr_rloc*, *dzrstrlmr_rloc*, *ddzasl*, *bpsdas_rloc*, *bpsdas*, *dzdvplmr*, *bzpdas_rloc*, *dzddvplmr*

Routines *gettofinish()*, *to_gather_rloc_on_rank0()*, *gettonext()*, *initialize_to()*, *getto()*

Needed modules

- *blocking* (*nfs()*, *lm2()*): Module containing blocking information

- *horizontal_data* (*dtheta1s()*, *dtheta1a()*, *dlh()*, *hdif_v()*, *costheta()*, *sintheta()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

- *parallel_mod*: This module contains the blocking information

- *radial_data* (*nrstart()*, *n_r_cmb()*, *nrstop()*)

- *truncation* (*l_max()*, *n_theta_maxstr()*, *n_r_maxstr()*, *n_phi_maxstr()*, *nrp()*): This module defines the grid points and the truncation

- *radial_functions* (*or4()*, *or2()*, *beta()*, *r()*, *orho1()*, *dbeta()*, *or3()*, *or1()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *logic* (*l_mag()*, *lverbose()*): Module containing the logicals that control the run

- *precision_mod*: This module controls the precision used in MagIC

- *physical_parameters* (*corfac()*, *ktopv()*, *kbotv()*): Module containing the physical parameters

- *lmloop_data* (*ulmmag()*, *llmmag()*)
- *legendre_grid_to_spec* (*legtfas2()*)
- *constants* (*two()*, *one()*): module containing constants and parameters used in the code.

Variables

- torsional_oscillations/**bpsdas** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bszas_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bspas_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzdvplmr_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzstrlmr** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzdvplmr** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzddvplmr** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bszas** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzddvplmr_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bspdas** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzlflmr_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bpzdas_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bzpdas** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzastrlmr_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**ddzasl** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**v2as** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzastrlmr** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bspdas_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzrstrlmr** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzcorlmr_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzlflmr** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzrstrlmr_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bpsdas_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bs2as_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzcorlmr** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bpzas** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bpzdas** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**dzstrlmr_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**v2as_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bpzas_rloc** (:,:) *[real,allocatable/public]*
- torsional_oscillations/**bzpdas_rloc** (:,:) *[real,allocatable/public]*

- torsional_oscillations/**bs2as** (:,:) *[real,allocatable/public]*

- torsional_oscillations/**bspas** (:,:) *[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_seq()`, *do_iteration_thetablocking_openmp()*

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_seq()`, *do_iteration_thetablocking_openmp()*

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_seq(), *do_iteration_thetablocking_openmp()*

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 *get_pas()*, *getpastr()*, *getastr()*

Needed modules

- *truncation* (*l_max()*): This module defines the grid points and the truncation
- *blocking* (*lm2()*): Module containing blocking information
- *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
- *constants* (*two()*, *one()*, *half()*): module containing constants and parameters used in the code.

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 *outto()*, *outpv()*

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 `out_TO.f90`

Quick access

Variables *rz*, *d_costf_initz*, *lfm*, *dvpm*, *vpm*, *zz*, *dplms*, *plms*, *strm*, *osints*, *corm*, *clm*, *astrm*, *rstrm*, *i_costf_initz*, *nzmaxs*, *lmmaxs*

Routines *outto()*, *initialize_outto_mod()*

Needed modules

- *blocking* (*sizethetab()*, *nfs()*, *nthetabs()*, *st_map()*): Module containing blocking information

- *communications* (*gt_oc()*, *gather_all_from_lo_to_rank0()*)

- *num_param* (*tscale()*): Module containing numerical and control parameters

- *chebint_mod* (*chebintinit()*, *chebint()*)

- *parallel_mod* (rank()): This module contains the blocking information

- *truncation* (*n_theta_max()*, *n_r_max()*, *l_max()*, *n_theta_maxstr()*, *lstressmem()*, *minc()*, *lm_max()*, *n_r_maxstr()*, *n_phi_max()*): This module defines the grid points and the truncation

- *cosine_transform* (*costf1()*)

- *charmanip* (*dble2str()*): This module contains several useful routines to manipule character strings

- *legendre_grid_to_spec* (*legtfas2()*, *legtfas()*)

- *plms_theta* (*plm_theta()*)

- *to_helpers* (*getpastr()*, *getastr()*, *get_pas()*): This module contains several helpful subroutines used in the TO calculations

- *output_data* (*zdens()*, *tag()*, *nzmaxa()*, *n_log_file()*, *log_file()*, *runid()*, *nsmaxa()*, *sdens()*): This module contains the parameters for output control

- *radial_functions* (*r_cmb()*, *i_costf_init()*, *drx()*, *r()*, *d_costf_init()*, *r_icb()*, *orho1()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *logic* (*l_save_out()*, *lverbose()*): Module containing the logicals that control the run

- *useful* (*logwrite()*): library with several useful subroutines

- *torsional_oscillations* (*dzstrlmr()*, *dzrstrlmr_rloc()*, *bszas()*, *v2as()*, *dzastrlmr()*, *bpzas()*, *dzddvplmr()*, *bspdas()*, *to_gather_rloc_on_rank0()*, *dzrstrlmr()*, *dzcorlmr()*, *dzdvplmr()*, *bzpdas()*, *bpzdas()*, *dzlflmr()*, *bs2as()*, *bspas()*, *bpsdas()*): This module contains information for TO calculation and output

- *precision_mod*: This module controls the precision used in MagIC

- *horizontal_data* (*gauss()*, *theta_ord()*, *phi()*, *sintheta()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

- *constants* (*two()*, *one()*, *four()*, *vol_oc()*, *pi()*, *half()*): module containing constants and parameters used in the code.

- *physical_parameters* (*pr()*, *lffac()*, *radratio()*, *ek()*, *ra()*, *prmag()*): Module containing the physical parameters

- *lmloop_data* (*llm()*, *ulm()*)

- *integration* (*rint_r()*): Radial integration functions

Variables

- `outto_mod`/**d_costf_initz** (:,:) *[real,private/allocatable]*

- `outto_mod`/**lmmaxs** *[integer,private]*

- `outto_mod`/**dplms** (:,:,:) *[real,private/allocatable]*

- `outto_mod`/**nzmaxs** (:) *[integer,private/allocatable]*

- `outto_mod`/**corm** (:,:) *[real,private/allocatable]*

- `outto_mod`/**lfm** (:,:) *[real,private/allocatable]*

- `outto_mod`/**astrm** (:,:) *[real,private/allocatable]*

- `outto_mod`/**dvpm** (:,:) *[real,private/allocatable]*

- `outto_mod`/**vpm** (:,:) *[real,private/allocatable]*

- `outto_mod`/**rstrm** (:,:) *[real,private/allocatable]*

- `outto_mod`/**osints** (:,:) *[real,private/allocatable]*

- `outto_mod`/**plms** (:,:,:) *[real,private/allocatable]*

- `outto_mod`/**i_costf_initz** (:,:) *[integer,private/allocatable]*

- `outto_mod`/**strm** (:,:) *[real,private/allocatable]*

- `outto_mod`/**zz** (:,:) *[real,private/allocatable]*

- `outto_mod`/**rz** (:,:) *[real,private/allocatable]*

- `outto_mod`/**clm** (:,:) *[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()`

10.17.9 `radial_spectra.f90`

Quick access

> **Routines** `rbpspec()`, `rbrspec()`

Needed modules

- `blocking` (`st_map()`): Module containing blocking information

- `num_param` (`escale()`): Module containing numerical and control parameters

- `horizontal_data` (`dLh()`): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

- `radial_data` (`n_r_icb()`)

- `output_data` (`tag()`): This module contains the parameters for output control

- `radial_functions` (`r_ic()`, `r_icb()`, `or2()`): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- `logic` (`l_cond_ic()`): Module containing the logicals that control the run

- `useful` (`cc2real()`): library with several useful subroutines

- `truncation` (`n_r_tot()`, `lm_max()`, `l_max()`, `n_r_ic_max()`, `n_r_max()`): This module defines the grid points and the truncation

- `precision_mod`: This module controls the precision used in MagIC

- `lmmapping` (`mappings()`)

- [constants](pi(), one(), four(), half()): module containing constants and parameters used in the code.

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()*

10.17.10 `Egeos.f90`

Quick access

> **Variables** *rhoz*, *orhoz*
>
> **Routines** *getegeos()*, *initialize_egeos_mod()*, *getdvptr()*

Needed modules

- *blocking* (*lm2l()*, *lm2mc()*, *lm2m()*): Module containing blocking information

- *communications* (*gt_oc()*, *gather_all_from_lo_to_rank0()*)

- *num_param* (*tscale()*): Module containing numerical and control parameters

- *horizontal_data* (*dlh()*, *dphi()*, *phi()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

- *physical_parameters* (*pr()*, *polind()*, *radratio()*, *g2()*, *strat()*, *ek()*, *ra()*, *g1()*, *prmag()*, *g0()*): Module containing the physical parameters

- *plms_theta* (*plm_theta()*)

- *output_data* (*zdens()*, *tag()*, *nzmaxa()*, *runid()*, *nsmaxa()*, *sdens()*): This module contains the parameters for output control

- *radial_functions* (*d_costf_init()*, *cheb_norm()*, *i_costf_init()*, *r_cmb()*, *r_icb()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *chebint_mod*

- *logic* (*l_anel()*, *l_corrmov()*, *lverbose()*): Module containing the logicals that control the run

- *truncation* (*n_phi_max()*, *n_r_max()*, *l_max()*, *minc()*, *n_m_max()*, *m_max()*, *nrpgeos()*, *lm_max()*, *n_r_maxgeos()*, *lm_maxgeos()*): This module defines the grid points and the truncation

- *precision_mod*: This module controls the precision used in MagIC

- *parallel_mod* (rank()): This module contains the blocking information

- *cosine_transform* (*costf1()*)

- *lmloop_data* (*llm()*, *ulm()*)

- *fft* (*fft_to_real()*): This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5

- *constants* (*two()*, *ci()*, *one()*, *half()*, *pi()*, *zero()*): module containing constants and parameters used in the code.

Variables

- egeos_mod/**orhoz** (:,:) *[real,private/allocatable]*

- egeos_mod/**d_costf_initz** (:,:) *[real,private/allocatable]*

- egeos_mod/**dplms** (:,:,:) *[real,private/allocatable]*

- egeos_mod/**osints** (:,:) *[real,private/allocatable]*

- egeos_mod/**nzmaxs** (:) *[integer,private/allocatable]*

- egeos_mod/**plms** (:,:,:) *[real,private/allocatable]*

- egeos_mod/**zz** (:,:) *[real,private/allocatable]*

- egeos_mod/**rhoz** (:,:) *[real,private/allocatable]*

- egeos_mod/**rz** (:,:) *[real,private/allocatable]*

- egeos_mod/**i_costf_initz** (:,:) *[integer,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()*

10.17.11 `outPV3.f90`

Quick access

Variables *plmz*, *vorold*, *dplmz*

Routines *initialize_outpv3()*, *getpvptr()*, *outpv()*

Needed modules

- *blocking* (*lm2l()*, *lm2mc()*, *lm2m()*, *lm2()*): Module containing blocking information

- *communications* (*gt_oc()*, *gather_all_from_lo_to_rank0()*)

- *horizontal_data* (*dlh()*, *dphi()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

- *physical_parameters* (*radratio()*): Module containing the physical parameters

---

- *to_helpers* (*getpastr()*): This module contains several helpful subroutines used in the TO calculations

- *plms_theta* (*plm_theta()*)

- *output_data* (*tag()*, *nzmaxa()*, *nsmaxa()*, *sdens()*): This module contains the parameters for output control

- *radial_functions* (*d_costf_init()*, *cheb_norm()*, *i_costf_init()*, *r_cmb()*, *r_icb()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *logic* (*l_sric()*, *lverbose()*): Module containing the logicals that control the run

- *truncation* (*n_phi_max()*, *n_r_max()*, *n_m_max()*, *nrp()*, *m_max()*, *minc()*, *lm_max()*, *l_max()*): This module defines the grid points and the truncation

- *precision_mod*: This module controls the precision used in MagIC

- *parallel_mod* (rank()): This module contains the blocking information

- *cosine_transform* (*costf1()*)

- *lmloop_data* (*llm()*, *ulm()*)

- *fft* (*fft_to_real()*): This file contains the subroutines called by fftJW: fft99a, fft99b, wpass2, wpass3, wpass4 and wpass5

- *constants* (*two()*, *ci()*, *one()*, *zero()*, *pi()*, *half()*): module containing constants and parameters used in the code.

Variables

- outpv3/**dplmz** (:,:,:) *[real,private/allocatable]*

- outpv3/**osints** (:,:) *[real,private/allocatable]*

- outpv3/**vorold** (:,:,:,:) *[real,private/allocatable]*

- outpv3/**plmz** (:,:,:) *[real,private/allocatable]*

- outpv3/**plms** (:,:,:) *[real,private/allocatable]*

- outpv3/**dplms** (:,:,:) *[real,private/allocatable]*

- outpv3/**rz** (:,:) *[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()*

10.17.12 `chebInt.f90`

Quick access

> **Routines** *chebintd()*, *chebint()*, *chebintinit()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *cosine_transform* (*costf1()*)
- *constants* (*two()*, *four()*, *half()*): module containing constants and parameters used in the code.
- *init_costf* (*init_costf1()*)
- *radial_der* (*get_dcheb()*): Radial derivatives functions
- *chebyshev_polynoms_mod* (*cheb_grid()*)

Variables

Subroutines and functions

subroutine chebint_mod/**chebintinit** (*zmin*, *zmax*, *znorm*, *nnorm*, *ngridpointsmax*, *z*, *ngridpoints*, *i_costf_init*, *d_costf_init*)

> **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]* :: dimension of z on input
> - **z** (ngridpointsmax) *[real,out]* :: grid points, dimension at >= nGridPointsMax
> - **ngridpoints** *[integer,out]* :: number of used grid points
> - **i_costf_init** (2 * ngridpointsmax + 2) *[integer,out]*
> - **d_costf_init** (2 * ngridpointsmax + 5) *[real,out]*
>
> **Called from** *outto()*, *getegeos()*
>
> **Call to** *cheb_grid()*, *init_costf1()*

function chebint_mod/**chebint** (*f*, *zmin*, *zmax*, *ngridpoints*, *ngridpointsmax*, *i_costf_init*, *d_costf_init*)

> **Parameters**
>
> - **f** (*) *[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
- **i_costf_init** (2 * ngridpointsmax + 2) *[integer,in]*
- **d_costf_init** (2 * ngridpointsmax + 5) *[real,in]*

Return **chebint** *[real]*

Called from *outto()*, *getegeos()*

function chebint_mod/**chebintd** (*f*, *lderiv*, *zmin*, *zmax*, *ngridpoints*, *ngridpointsmax*, *i_costf_init*, *d_costf_init*)

Parameters

- **f** (*) *[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
- **i_costf_init** (2 * ngridpointsmax + 2) *[integer,in]*
- **d_costf_init** (2 * ngridpointsmax + 5) *[real,in]*

Return **chebintd** *[real]*

Called from *getegeos()*

10.17.13 `outOmega.f90`

Description

This module allows to compute the axisymmetric zonal flow versus the cylindrical radius s. By

Quick access

Variables *nsmax*

Routines *lnpas2tr()*, *outomega()*

Needed modules

- *blocking* (*lm2()*): Module containing blocking information
- *output_data* (*tag()*): This module contains the parameters for output control
- *precision_mod*: This module controls the precision used in MagIC
- *cosine_transform* (*costf1()*)
- *plms_theta* (*plm_theta()*)
- *truncation* (*n_r_max()*, *l_max()*, *minc()*, *lm_max()*): This module defines the grid points and the truncation

---

- *radial_functions* (*d_costf_init()*, *r_cmb()*, *i_costf_init()*, *r_icb()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *logic* (*lverbose()*): Module containing the logicals that control the run

- *constants* (*two()*, *one()*, *half()*): module containing constants and parameters used in the code.

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 *output()*

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()*

10.17.14 `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_fluxes()*, *get_perppar()*, *get_nlblayers()*, *get_helicity()*

Needed modules

- *blocking* (*sizethetab()*, *nfs()*): Module containing blocking information
- *legendre_grid_to_spec* (*legtfas2()*, *legtfas()*)
- *precision_mod*: This module controls the precision used in MagIC
- *horizontal_data* (*osn2()*, *sn2()*, *o_sin_theta_e2()*, *costheta()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *radial_data* (*n_r_cmb()*, *n_r_icb()*)
- *physical_parameters* (*ek()*, *vischeatfac()*): Module containing the physical parameters
- *truncation* (*n_phi_max()*, *l_maxmag()*, *l_max()*, *nrp()*): This module defines the grid points and the truncation
- *radial_functions* (*temp0()*, *visc()*, *orho2()*, *beta()*, *orho1()*, *or2()*, *or1()*, *or4()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *logic* (*l_mag_nl()*): Module containing the logicals that control the run
- *constants* (*two()*, *pi()*, *one()*, *third()*, *half()*): module containing constants and parameters used in the code.

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_seq()`, *do_iteration_thetablocking_openmp()*

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_seq()`, *do_iteration_thetablocking_openmp()*

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*, *fvisclmr*, *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_seq()`, *[do_iteration_thetablocking_openmp()](#)*

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_seq(), *do_iteration_thetablocking_openmp()*

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** *mapdatamag()*, *getlm2lmo()*, *mapdatahydro()*, *readstartfields()*, *mapdatar()*

Needed modules

- *blocking* (*lm2l()*, *lmstartb()*, *lm2m()*, *nlmbs()*, *lmstopb()*, *lm2()*): Module containing blocking information

- *precision_mod*: This module controls the precision used in MagIC

- *cosine_transform* (*costf1()*)

- *physical_parameters* (*pr()*, *sigma_ratio()*, *ktopv()*, *radratio()*, *kbotv()*, *ek()*, *ra()*, *prmag()*): Module containing the physical parameters

- *radial_data* (*n_r_cmb()*, *n_r_icb()*)

- *init_costf* (*init_costf1()*)

- *truncation* (*n_r_ic_maxmag()*, *n_r_max()*, *l_max()*, *lm_maxmag()*, *n_r_maxmag()*, *m_max()*, *n_phi_tot()*, *lm_max()*, *lmagmem()*, *minc()*, *n_r_ic_max()*, *nalias()*): This module defines the grid points and the truncation

- *radial_functions* (*d_costf1_ic_init()*, *i_costf_init()*, *r()*, *d_costf_init()*, *cheb_norm()*, *i_costf1_ic_init()*, *cheb_norm_ic()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *logic* (*l_rot_ma()*, *l_mag_lf()*, *l_mag()*, *l_rot_ic()*, *l_heat()*, *l_sric()*, *l_cond_ic()*, *l_srma()*): Module containing the logicals that control the run

- *init_fields* (*tomega_ic1()*, *omega_ic2()*, *n_start_file()*, *tomega_ma2()*, *scale_b()*, *scale_s()*, *tshift_ic2()*, *omega_ma2()*, *scale_v()*, *tomega_ma1()*, *tshift_ma2()*, *omegaosz_ic1()*, *start_file()*, *inform()*, *tshift_ic1()*, *tshift_ma1()*, *omega_ic1()*, *tipdipole()*, *omegaosz_ma1()*, *omegaosz_ma2()*, *omega_ma1()*, *omegaosz_ic2()*, *tomega_ic2()*)

- *constants* (*zero()*, *two()*, *c_z10_omega_ic()*, *pi()*, *c_z10_omega_ma()*): module containing constants and parameters used in the code.

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()*

Call to *init_costf1()*

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

- *output_data* (*n_rst_file()*, *rst_file()*): This module contains the parameters for output control
- *num_param* (*tscale()*): Module containing numerical and control parameters
- *precision_mod*: This module controls the precision used in MagIC
- *physical_parameters* (*pr()*, *sigma_ratio()*, *radratio()*, *ek()*, *ra()*, *prmag()*): Module containing the physical parameters
- *truncation* (*n_theta_max()*, *n_r_max()*, *lm_maxmag()*, *l_max()*, *n_phi_tot()*, *lm_max()*, *nalias()*, *minc()*, *n_r_ic_max()*, *n_r_maxmag()*, *n_r_ic_maxmag()*): This module defines the grid points and the truncation
- *logic* (*l_mag()*, *l_cond_ic()*, *l_heat()*): Module containing the logicals that control the run
- *fieldslast* (*d_omega_ic_dtlast()*, *lorentz_torque_malast()*, *d_omega_ma_dtlast()*, *lorentz_torque_iclast()*): This module contains time-derivaties array of the previous time-step They are needed in the time-stepping scheme.
- *init_fields* (*inform()*, *tomega_ic1()*, *omega_ic2()*, *omega_ic1()*, *tomega_ma2()*, *omegaosz_ma1()*, *omegaosz_ma2()*, *omega_ma1()*, *omega_ma2()*, *tomega_ma1()*, *omegaosz_ic1()*, *omegaosz_ic2()*, *tomega_ic2()*)

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** *random()*, *l_correct_step()*, *cc22real()*, *cc2real()*, *safeopen()*, *factorise()*, *safeclose()*, *logwrite()*

Needed modules

- *output_data* (*log_file()*, *n_log_file()*): 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
- *constants* (*two()*, *one()*, *half()*): module containing constants and parameters used in the code.
- *parallel_mod* (rank()): This module contains the blocking information

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 *inits()*, *initv()*, *initb()*

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_costf2()*, *init_costf1()*, *init_fft()*

function useful/**cc2real**(*c*, *m*)

> **Parameters**
>
> > - **c** *[complex,in]*
> > - **m** *[integer,in]*
>
> **Return** **cc2real** *[real]*
>
> **Called from** *spectrum_average()*, *hint2tor()*, *hint2dpol()*, *getstartfields()*, *hint2pol()*, *get_u_square()*, *getdlm()*, *get_power()*, *spectrum_temp()*, *get_poltorrms()*, *outpar()*, *spectrum_temp_average()*, *rbpspec()*, *spectrum()*, *get_e_mag()*, *get_e_kin()*, *rbrspec()*

function useful/**cc22real**(*c1*, *c2*, *m*)

> **Parameters**
>
> > - **c1** *[complex,in]*
> > - **c2** *[complex,in]*
> > - **m** *[integer,in]*
>
> **Return** **cc22real** *[real]*
>
> **Called from** *get_power()*, *spectrum()*, *get_e_mag()*

subroutine useful/**safeopen**(*nf*, *file_name*)

> **Parameters**
>
> > - **nf** *[integer,in]*
> > - **file_name** *[character,in]*
>
> **Called from** *spectrum_average()*, *step_time()*, *output()*, *spectrum_temp_average()*, *lmloop()*

subroutine useful/**safeclose**(*nf*)

> **Parameters** **nf** *[integer,in]*

Called from *spectrum_average()*, *step_time()*, *output()*, *spectrum_temp_average()*, *lmloop()*

subroutine useful/**logwrite**(*message*)

> **Parameters message** *[character,in]*
>
> **Called from** *outto()*, *dt_courant()*, *getstartfields()*, *step_time()*, *output()*, *precalc()*, *initialize_blocking()*, *initialize_movie_data()*

10.19.2 `char_manip.f90`

Description

This module contains several useful routines to manipule character strings

Quick access

> **Routines** *length_to_blank()*, *length_to_char()*, *str2dble()*, *capitalize()*, *delete_string()*, *dble2str()*

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** *step_time()*, *readnamelists()*, *get_movie_type()*

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()*, *writenamelists()*, *readnamelists()*, *get_movie_type()*

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 *outto()*, *step_time()*, *output()*

10.19.3 `hdf5Helpers.f90`

Description

This module contains several useful tools to manipulate HDF5 files

Quick access

Variables *writehdf5_attribute*, *readhdf5_attribute*

Routines *readhdf5_attr_int()*, *write_dataset()*, *writehdf5_attr_dble()*, *writehdf5_attr_int()*, *readhdf5_attr_dble()*

Needed modules

- *blocking* (*st_map()*, *lo_map()*): Module containing blocking information

- *lmloop_data* (*llm()*, *ulm()*)

- *precision_mod*: This module controls the precision used in MagIC

- hdf5

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]*

m

M

P

U

V