Documentation of MagIC

Release 6.0

The MagIC dev team

Feb 10, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Foreword

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

MagIC uses Chebyshev polynomials or finite difference in the radial direction and spherical harmonic decomposition in the azimuthal and latitudinal directions. MagIC supports several Implicit-Explicit time schemes where the nonlinear terms and the Coriolis force are treated explicitly, while the remaining linear terms are treated implicitly.

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.

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)
- Benchmark for double diffusive convection: Breuer et al. (2010, GJI, 183, 150-162)
- Anelastic benchmark: Jones et al. (2011, Icarus, 216, 120-135)
- In case you use the SHTns library for the spherical harmonics transforms (MagIC 5.3 or later), please also cite: Schaeffer (2013, GGG, 14, 751-758)

See also:

A (tentative) comprehensive list of the publications that have been produced to date (May 2019) using **MagIC** is accessible here. To date, more than **100 publications** have been-accepted in more than 10 different peer-reviewed journals: PEPI (22), Icarus (11), E&PSL (7), GJI (8), A&A (6), GRL (4), JFM (6), GAFD (3), Nature (2), etc.

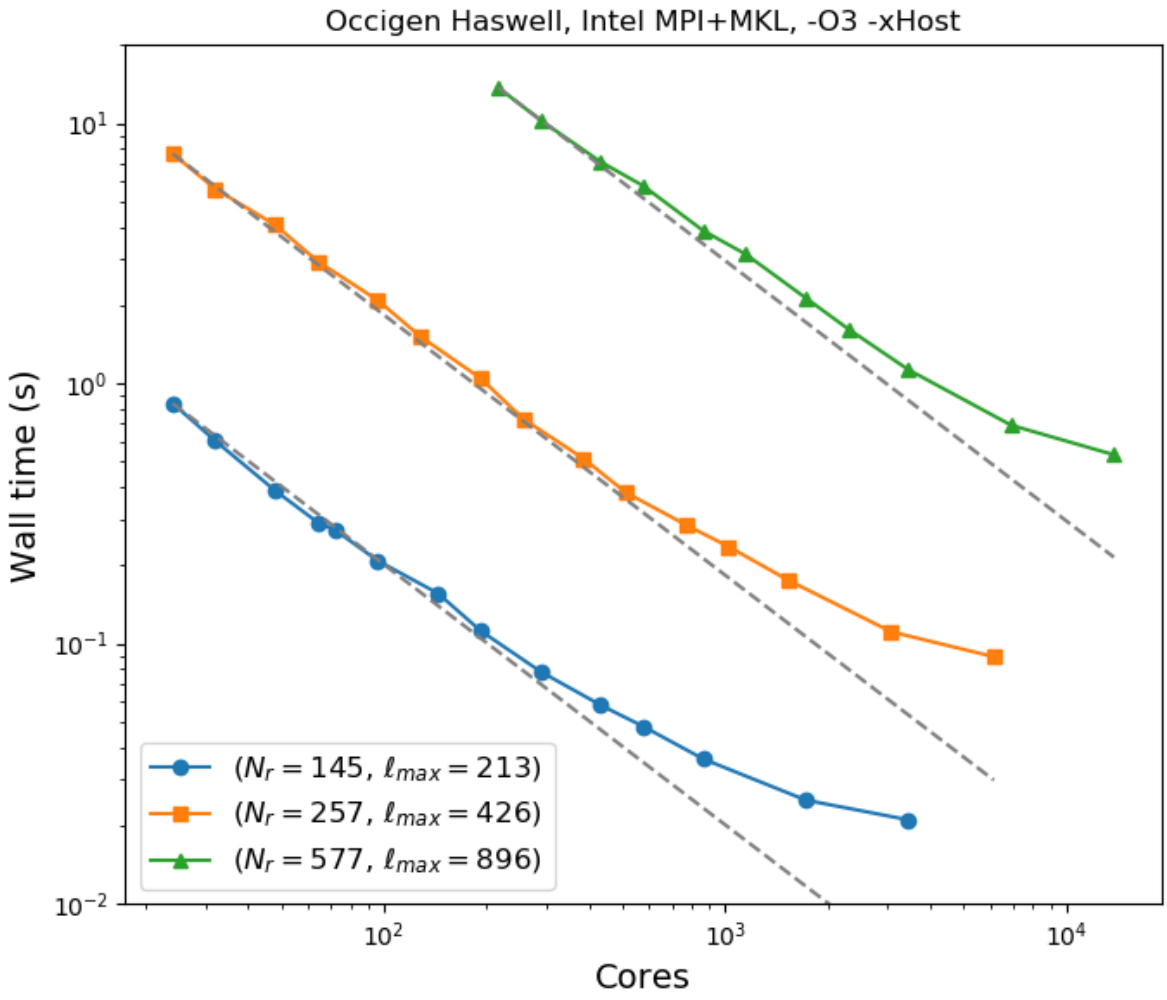

Fig. 1: Mean walltime of the MagIC code on the supercomputer Occigen versus number of cores for a Boussinesq dynamo model computed at three different numerical resolutions $(N_r, \ell_{\max})$. The dashed grey lines show the ideal scalings.

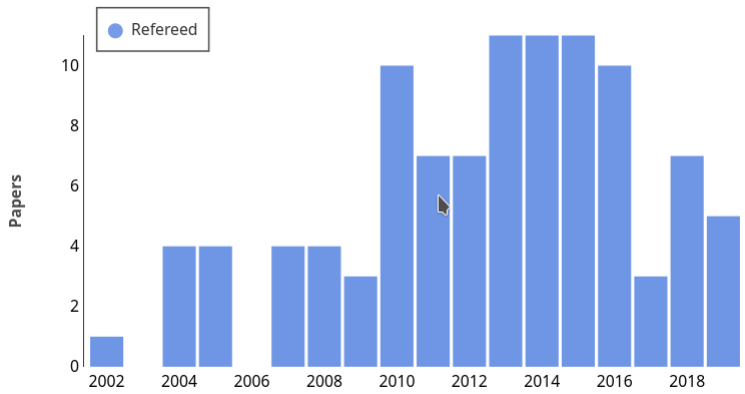

Fig. 2: Number of peer-reviewed publications produced using **MagIC**

CHAPTER

TWO

GET MAGIC AND RUN IT

2.1 Download the code

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

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

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

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

2.2 Setting up the environment variables

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

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

```
$ source sourceme.sh
```

In case you use csh or tcsh, rather use

```
$ source sourceme.csh
```

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

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

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

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

To get started, you then need to compile the code.

2.3 Install SHTns (recommended)

SHTns is a an open-source library for the Spherical Harmonics transforms. It is significantly faster than the native transforms implemented in MagIC, and it is hence **recommended** (though not mandatory) to install it. To install the library, first define a C compiler

```
$ export CC= gcc
```

or

```
$ export CC= icc
```

Then make sure a FFT library such FFTW or the MKL is installed on the target machine. Then make use of the install script

```
cd $MAGIC_HOME/bin
./install-shtns.sh
```

or install it manually after downloading and extracting the latest version here

```
./configure --enable-openmp --enable-ishioka --enable-magic-layout --prefix=$HOME/local
```

if FFTW is used or

```
./configure --enable-openmp --enable-ishioka --enable-magic-layout --prefix=$HOME/local -
↪-enable-mkl
```

if the MKL is used. Possible additional options may be required depending on the machine (check the website). Then compile and install the library

```
make
make install
```

2.4 Setting up compiler options and compiling

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

2.4.1 Generic compiling options

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

- `ARCH` Set it to '64' for 64 bit architecture or to '32' for 32 bit architecture
- `PRECISION` Set it to 'dble' for double-precision calculations or to 'sngl' for single-precision calculations
- `OUT_PREC` Set it to 'dble' for double-precision in binary outputs or to 'sngl' for single precision
- `USE_MPI` Set to `yes` to use MPI, set it to `no` if you want a serial version of the code .
- `USE_OMP` Set it to `yes` to use the hybrid version of the code, or to `no` for a pure MPI (or serial) version.
- `USE_PRECOND` Set to `yes` to perform some pre-conditioning of the matrices.

- `USE_FFTLIB` This option lets you select the library you want to use for Fast Fourier Transforms. This can be set to 'JW', 'FFTW' or 'MKL'. 'JW' refers to the built-in library by **J** ohannes **W** icht, FFTW refers to the Fastest Fourier Transform in the West, while 'MKL' refers to the Intel Math Kernel Library. Use 'JW' if you don't have Intel MKL installed.
- `USE_DCTLIB` This option lets you select the library you want to use for Discrete Cosine Transforms. This can be set to 'JW', 'FFTW' or 'MKL'.
- `USE_LAPACKLIB` This option allows you to select the library you want to use for LU factorisation. This can be set to 'JW', 'MKL', 'LIBFLAME' or 'LAPACK'. 'LIBFLAME' refers to the AMD dense matrix solvers libflame.
- `USE_SHTNS` Set to `yes` to use SHTns library for spherical harmonics transforms. The helper script `install-shtns.sh` is available in the `bin` directory to help installing SHTns.
- `CMAKE_BUILD_TYPE` Set to `Debug` to enable the full debug flags.

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

2.4.2 Using `CMake` (recommended)

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

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

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

```
$ export FC=mpiifort
$ export CC=icc
```

for bash/ksh/zsh users and

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

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

```
$ cmake .. -DUSE_SHTNS=yes
```

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

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

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

```
$ make -j
```

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

If you want to recompile the code from scratch do

```
$ make clean
```

to remove all the files generated by the compiler.

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

2.4.3 Using `make` (backup solution)

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

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

Select compiler

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

List of default compilers

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

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

Select compiling options

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

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

MPI_INCPATH

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

Other compilers

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

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

```
$ make -j
```

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

If you want to recompile the code from scratch do

```
$ make clean
```

to remove all the files generated by the compiler.

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

2.5 Preparing a production run

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

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

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

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

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

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

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

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

CHAPTER

THREE

FORMULATION OF THE (MAGNETO)-HYDRODYNAMICS PROBLEM

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

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

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

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

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

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

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

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

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

describes density variations caused by variations in temperature T, pressure p, and composition ξ. The latter contribution needs to be considered for describing the effects of light elements released from a growing solid iron core in a so-called double diffusive approach.

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

$$\rho T\left(\frac{\partial s}{\partial t}+\boldsymbol{u}\cdot\boldsymbol{\nabla}s\right)=\boldsymbol{\nabla}\cdot(k\boldsymbol{\nabla}T)+\Phi_\nu+\frac{\lambda}{\mu_0}\left(\boldsymbol{\nabla}\times\boldsymbol{B}\right)^2+\epsilon, \tag{3.4}$$

where Φ_ν corresponds to the viscous heating expressed by

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

Note that we use here the summation convention over the indices i and j. The second last term on the right hand side is the Ohmic heating due to electric currents. The last term is a volumetric sink or source term that can describe various

effects, for example radiogenic heating, the mixing-in of the light elements or, when radially dependent, potential variations in the adiabatic gradient (see below). For chemical composition, we finally use

$$\rho\left(\frac{\partial \xi}{\partial t}+\boldsymbol{u}\cdot\boldsymbol{\nabla}\xi\right)=\boldsymbol{\nabla}\cdot(k_\xi\boldsymbol{\nabla}\xi)+\epsilon_\xi, \tag{3.5}$$

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

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

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

$$\frac{\partial \boldsymbol{B}}{\partial t}=\boldsymbol{\nabla}\times(\boldsymbol{u}\times\boldsymbol{B})+\lambda\boldsymbol{\Delta}\boldsymbol{B}. \tag{3.7}$$

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

$$\alpha=-\frac{1}{\rho}\left(\frac{\partial \rho}{\partial T}\right)_{p,\xi}, \tag{3.8}$$

the compressibility at constant temperature

$$\beta=\frac{1}{\rho}\left(\frac{\partial \rho}{\partial p}\right)_{T,\xi}$$

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

$$\delta=-\frac{1}{\rho}\left(\frac{\partial \rho}{\partial \xi}\right)_{p,T}, \tag{3.9}$$

3.1 The reference state

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

$$\epsilon\sim\frac{T'}{\tilde{T}}\sim\frac{p'}{\tilde{p}}\sim\frac{\rho'}{\tilde{\rho}}\sim\ldots\ll 1. \tag{3.10}$$

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

$$\nabla\tilde{p}=\tilde{\rho}\tilde{\boldsymbol{g}}. \tag{3.11}$$

Convective motions are supposed to be strong enough to provide homogeneous entropy (and composition). The reference state is thus adiabatic and its gradients can be expressed in terms of the pressure gradient (3.11):

$$\frac{\nabla\tilde{T}}{\tilde{T}}=\frac{1}{\tilde{T}}\left(\frac{\partial T}{\partial p}\right)_s\nabla p=\frac{\alpha}{c_p}\tilde{\boldsymbol{g}}, \tag{3.12}$$

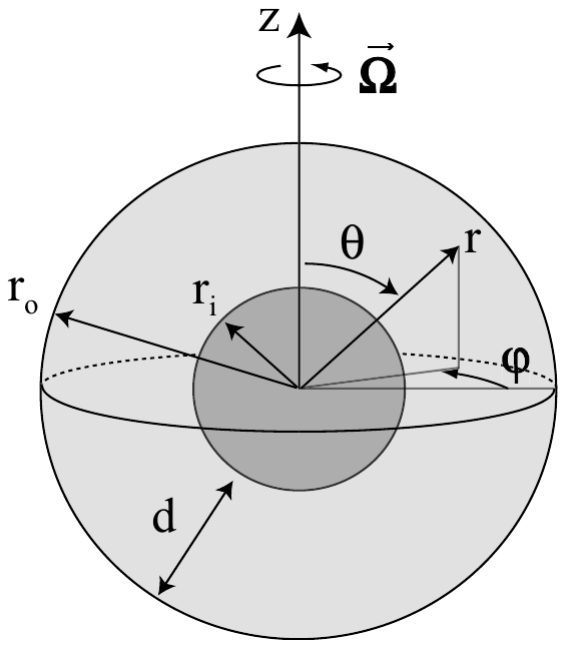

Fig. 1: Sketch of the spherical shell model and its system of coordinate.

$$\frac{\nabla\tilde{\rho}}{\tilde{\rho}} = \frac{1}{\tilde{\rho}}\left(\frac{\partial\rho}{\partial p}\right)_s \nabla p = \beta\tilde{\rho}\tilde{\boldsymbol{g}}. \tag{3.13}$$

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

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

and

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

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

$$\frac{\alpha\nabla T}{\beta\nabla\rho} \approx \alpha\tilde{T}\frac{Di}{Co}. \tag{3.14}$$

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

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

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

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

Square brackets denote order of magnitude estimates here. We have used the fact that the reference state is static and assume time scale of changes are comparable (or slower) ρ' than the time scales represented by u and that length scales

associated to the gradient operator are not too small. We can then neglect local variations in ρ' which means that sound waves are filtered out. This first order continuity equation thus simply reads:

$$\boldsymbol{\nabla} \cdot (\tilde{\rho}\boldsymbol{u}) = 0. \tag{3.15}$$

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

3.2 Boussinesq approximation

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

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

When using typical number for Earth, (3.14) becomes 0.05 so that pressure effects on density may be neglected. The first order Navier-Stokes equation (after to zero order hydrostatic reference solution has been subtracted) then reads:

$$\tilde{\rho}\left(\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \boldsymbol{\nabla}\,\boldsymbol{u}\right) = -\boldsymbol{\nabla}p' - 2\rho\boldsymbol{\Omega} \times \boldsymbol{u} + \alpha\tilde{g}_o T'\frac{\boldsymbol{r}}{r_o} + \delta\tilde{g}_o\xi'\frac{\boldsymbol{r}}{r_o} + \frac{1}{\mu_0}(\boldsymbol{\nabla} \times \boldsymbol{B}) \times \boldsymbol{B} + \tilde{\rho}\nu\Delta\boldsymbol{u}. \tag{3.16}$$

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

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

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

The first order energy equation becomes

$$\tilde{\rho}\left(\frac{\partial T'}{\partial t} + \boldsymbol{u} \cdot \boldsymbol{\nabla}T'\right) = \kappa\Delta T' + \epsilon, \tag{3.17}$$

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

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

defined the thermal diffusivity

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

and adjusted the definition of ϵ. Finally the first order equation for chemical composition becomes

$$\tilde{\rho}\left(\frac{\partial \xi'}{\partial t} + \boldsymbol{u} \cdot \boldsymbol{\nabla}\xi'\right) = \kappa_\xi\Delta\xi' + \epsilon_\xi, \tag{3.18}$$

where we have assumed a homogeneous k_ξ and adjusted the definition of ϵ_ξ.

MagIC solves a dimensionless form of the differential equations. Time is scaled in units of the viscous diffusion time d^2/ν, length in units of the shell thickness d, temperature in units of the temperature drop $\Delta T = T_o - T_i$ over the

shell, composition in units of the composition drop $\Delta\xi = \xi_o - \xi_i$ over the shell and magnetic field in units $(\mu\lambda\tilde{\rho}\Omega)^{1/2}$. Technically the transition to the dimensionless form is achieved by the substitution

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

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

$$\left(\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u}\cdot\boldsymbol{\nabla}\,\boldsymbol{u}\right) = -\boldsymbol{\nabla}p' - \frac{2}{E}\boldsymbol{e_z}\times\boldsymbol{u} + \frac{Ra}{Pr}T'\frac{\boldsymbol{r}}{r_o} + \frac{Ra_\xi}{Sc}\xi'\frac{\boldsymbol{r}}{r_o} + \frac{1}{EPm}(\boldsymbol{\nabla}\times\boldsymbol{B})\times\boldsymbol{B} + \Delta\boldsymbol{u}, \tag{3.19}$$

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

3.3 Anelastic approximation

The anelastic approximation adopts the simplified continuity (3.15). The background state can be specified in different ways, for example by providing profiles based on internal models and/or ab initio simulations. We will assume a polytropic ideal gas in the following.

3.3.1 Analytical solution in the limit of an ideal gas

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

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

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

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

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

Warning:

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

3.3.2 Anelastic MHD equations

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

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

This leads to the following sets of dimensionless equations:

$$\left(\frac{\partial \boldsymbol{u}}{\partial t}+\boldsymbol{u}\cdot\boldsymbol{\nabla}\boldsymbol{u}\right)=-\boldsymbol{\nabla}\left(\frac{p'}{\tilde{\rho}}\right)-\frac{2}{E}\boldsymbol{e_z}\times\boldsymbol{u}+\frac{Ra}{Pr}\tilde{g}\,s'\,\boldsymbol{e_r}+\frac{Ra_\xi}{Sc}\tilde{g}\,\xi'\,\boldsymbol{e_r}+\frac{1}{Pm\,E\,\tilde{\rho}}\left(\boldsymbol{\nabla}\times\boldsymbol{B}\right)\times\boldsymbol{B}+\frac{1}{\tilde{\rho}}\boldsymbol{\nabla}\cdot\mathsf{S}, \tag{3.20}$$

$$\boldsymbol{\nabla}\cdot\tilde{\rho}\boldsymbol{u}=0, \tag{3.21}$$

$$\boldsymbol{\nabla}\cdot\boldsymbol{B}=0, \tag{3.22}$$

$$\tilde{\rho}\left(\frac{\partial \xi'}{\partial t}+\boldsymbol{u}\cdot\boldsymbol{\nabla}\xi'\right)=\frac{1}{Sc}\boldsymbol{\nabla}\cdot\left(\kappa_\xi(r)\tilde{\rho}\boldsymbol{\nabla}\xi'\right) \tag{3.23}$$

$$\frac{\partial \boldsymbol{B}}{\partial t}=\boldsymbol{\nabla}\times\left(\boldsymbol{u}\times\boldsymbol{B}\right)-\frac{1}{Pm}\boldsymbol{\nabla}\times\left(\lambda(r)\,\boldsymbol{\nabla}\times\boldsymbol{B}\right). \tag{3.24}$$

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

3.3.3 Entropy equation and turbulent diffusion

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

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

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

$$\epsilon\frac{T'/\delta^2}{T/d^2}, \tag{3.25}$$

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 (3.25) will be much smaller than unity. The second terms will thus effectively acts as a radial-dependent heat source or sink that will drive or hinder convection. This is one of the physical motivation to rather introduce a **turbulent diffusivity** that will be approximated by

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

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

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

$$\tilde{\rho}\tilde{T}\left(\frac{\partial s'}{\partial t}+\boldsymbol{u}\cdot\boldsymbol{\nabla}s'\right)=\frac{1}{Pr}\boldsymbol{\nabla}\cdot\left(\kappa(r)\tilde{\rho}\tilde{T}\boldsymbol{\nabla}s'\right)+\frac{Pr\,Di}{Ra}\Phi_\nu+\frac{Pr\,Di}{Pm^2\,E\,Ra}\lambda(r)\left(\boldsymbol{\nabla}\times\boldsymbol{B}\right)^2, \tag{3.26}$$

A comparison with (3.20) reveals meaning of the different non-dimensional numbers that scale viscous and Ohmic heating. The fraction Pr/Ra simply expresses the ratio of entropy and flow in the Navier-Stokes equation, while the additional factor $1/EPm$ reflects the scale difference of magnetic field and flow. Then remaining dissipation number Di then expresses the relative importance of viscous and Ohmic heating compared to buoyancy and Lorentz force in the Navier-Stokes equation. For small Di both heating terms can be neglected compared to entropy changes due to advection, an limit that is used in the Boussinesq approximation.

3.4 Dimensionless control parameters

The equations (3.20)-(3.26) are governed by four dimensionless numbers: the Ekman number

$$E=\frac{\nu}{\Omega d^2}, \tag{3.27}$$

the thermal Rayleigh number

$$Ra=\frac{\alpha_o g_o T_o d^3\Delta s}{c_p\kappa_o\nu_o}, \tag{3.28}$$

the compositional Rayleigh number

$$Ra_\xi=\frac{\delta_o g_o d^3\Delta\xi}{\kappa_\xi\nu_o}, \tag{3.29}$$

the Prandtl number

$$Pr=\frac{\nu_o}{\kappa_o}, \tag{3.30}$$

the Schmidt number

$$Sc=\frac{\nu_o}{\kappa_\xi}, \tag{3.31}$$

and the magnetic Prandtl number

$$Pm=\frac{\nu_o}{\lambda_i}. \tag{3.32}$$

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

$$\eta=\frac{r_i}{r_o}, \tag{3.33}$$

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

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

$$Ra=\frac{\alpha g_o d^3\Delta T}{\kappa\nu}. \tag{3.34}$$

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

See also:

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

3.4.1 Usual diagnostic quantities

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

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

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

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

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

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

a measure for the ratio of inertial to Coriolis forces.

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

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

See also:

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

3.5 Boundary conditions and treatment of inner core

3.5.1 Mechanical conditions

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

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

See also:

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

3.5.2 Magnetic boundary conditions and inner core conductivity

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

Depending on the physical problem you want to model, treating the inner core as an insulator is not realistic either, and it might instead be more appropriate to assume that it has the same electrical conductivity as the fluid shell. In this case, an equation equivalent to (3.24) must be solved for the inner core, where the velocity field simply describes the solid body rotation of the inner core with respect to the reference frame. At the inner core boundary a continuity condition for the magnetic field and the horizontal component of the electrical field apply.

See also:

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

3.5.3 Thermal boundary conditions and distribution of buoyancy sources

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

See also:

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

3.5.4 Chemical composition boundary conditions

They are treated in a very similar manner as the thermal boundary conditions

See also:

The boundary conditions for composition can be adjusted with the parameters *ktopxi* and *kbotxi* in the *&phys_param* section of the input namelist.

CHAPTER

FOUR

NUMERICAL TECHNIQUE

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

4.1 Poloidal/toroidal decomposition

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

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

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

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

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

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

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

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

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

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

$$\tilde{\rho}\boldsymbol{u} = -(\Delta_H W)\, \boldsymbol{e_r} + \left(\frac{1}{r} \frac{\partial^2 W}{\partial r \partial \theta} + \frac{1}{r \sin\theta} \frac{\partial Z}{\partial \phi} \right) \boldsymbol{e_\theta} + \left(\frac{1}{r \sin\theta} \frac{\partial^2 W}{\partial r \partial \phi} - \frac{1}{r} \frac{\partial Z}{\partial \theta} \right) \boldsymbol{e_\phi}, \tag{4.4}$$

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

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

Using the horizontal part of the divergence operator

$$\boldsymbol{\nabla}_H = \frac{1}{r\sin\theta}\frac{\partial}{\partial\theta}\sin\theta\,\boldsymbol{e}_\theta + \frac{1}{r\sin\theta}\frac{\partial}{\partial\phi}\,\boldsymbol{e}_\phi$$

above expressions can be simplified to

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

and

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

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

4.2 Spherical harmonic representation

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

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

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

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

This means that

$$Y_\ell^m(\theta,\phi) = \left(\frac{(2\ell+1)}{4\pi}\frac{(\ell-|m|)!}{(\ell+|m|)!}\right)^{1/2}P_\ell^m(\cos\theta)\,e^{im\phi},$$

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

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

with

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

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

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

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

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

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

See also:

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

4.2.1 Special recurrence relations

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

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

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

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

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

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

where c_ℓ^m is defined by

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

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

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

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

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

so that

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

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

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

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

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

See also:

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

The second operator used to formulate colatitude derivatives is

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

The action of this operator on the Legendre polynomials reads

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

so that

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

See also:

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

The third combined operator is defined by:

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

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

Acting with ϑ_3 on a Legendre function gives:

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

which results into:

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

See also:

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

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

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

Acting with ϑ_3 on a Legendre function gives:

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

which results into:

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

See also:

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

4.3 Radial representation

In MagIC, the radial dependencies are either expanded into complete sets of functions, the Chebyshev polynomials $\mathcal{C}(x)$; or discretized using finite differences. For the former approach, the Chebyshev polynomial of degree n is defined by

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

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

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

with

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

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

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

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

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

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

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

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

This particular choice has two advantages. For one, the grid points become denser toward the inner and outer radius and better resolve potential thermal and viscous boundary layers. In addition, type I Discrete Cosine Transforms (DCTs) can be employed to switch between grid representation (4.16) and Chebyshev representations (4.17), rendering this procedure a fast-Chebyshev transform.

See also:

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

4.4 Spectral equations

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

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

4.4.1 Equation for the poloidal potential W

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

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

For the viscosity term, one gets

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

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

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

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

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

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

See also:

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

4.4.2 Equation for the toroidal potential Z

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

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

The pressure gradient, one has

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

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

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

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

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

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

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

See also:

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

4.4.3 Equation for pressure P

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

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

This relates to the total divergence via:

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

The time-derivative term is thus expressed by

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

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

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

The viscosity term then reads

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

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

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

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

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

See also:

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

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

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

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

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

4.4.4 Equation for entropy s

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

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

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

See also:

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

4.4.5 Equation for chemical composition ξ

The equation for the chemical composition is given by

$$\begin{aligned} &\frac{1}{Sc}\left[\left(Sc\frac{\partial}{\partial t}+\kappa_\xi\frac{\ell(\ell+1)}{r^2}\right)\mathcal{C}_n\right. \\ &\quad -\kappa_\xi\left(\frac{d\ln\kappa_\xi}{dr}+\frac{d\ln\tilde{\rho}}{dr}++\frac{2}{r}\right)\mathcal{C}'_n \\ &\quad \left.-\kappa_\xi\,\mathcal{C}''_n\right]\quad \xi_{\ell mn} \\ &=\mathcal{N}^\xi_{\ell m}=\int d\Omega\,Y_\ell^{m\star}\,\mathcal{N}^\xi=\int d\Omega\,Y_\ell^{m\star}\,\left[-\boldsymbol{u}\cdot\boldsymbol{\nabla}\xi\right]. \end{aligned} \tag{4.25}$$

Once again, the numerical evaluation of the right-hand-side (i.e. the non-linear term) is discussed *below*.

See also:

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

4.4.6 Equation for the poloidal magnetic potential g

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

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

The spectral form then reads

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

See also:

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

4.4.7 Equation for the toroidal magnetic potential h

The equation for the toroidal magnetic field coefficient reads

$$\begin{aligned} &\frac{\ell(\ell+1)}{r^2}\left[\left(\frac{\partial}{\partial t}+\frac{1}{Pm}\lambda\frac{\ell(\ell+1)}{r^2}\right)\mathcal{C}_n\right. \\ &\quad -\frac{1}{Pm}\frac{d\lambda}{dr}\mathcal{C}'_n \\ &\quad \left.-\frac{1}{Pm}\lambda\,\mathcal{C}''_n\right]\quad h_{\ell mn} \\ &=\mathcal{N}^h_{\ell m}=\int d\Omega\,Y_\ell^{m\star}\,\mathcal{N}^h=\int d\Omega\,Y_\ell^{m\star}\,\boldsymbol{e_r}\cdot(\boldsymbol{\nabla}\times\boldsymbol{D})\,. \end{aligned} \tag{4.27}$$

See also:

The exact computation of the linear terms of (4.27) are coded in the subroutines *get_bMat*

Note: We note that the terms on the left hand side of (4.26) and (4.27) 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:

$$\boldsymbol{D} = \boldsymbol{\nabla} \times (\boldsymbol{u} \times \boldsymbol{B}) \ . \tag{4.28}$$

We have now derived a full set of equations (4.20), (4.21), (4.22), (4.24), (4.26) and (4.27), each describing the evolution of a single spherical harmonic mode of the six unknown fields (assuming that the terms on the right hand side are given). Each equation couples $N+1$ Chebyshev coefficients for a given spherical harmonic mode (ℓ, m). Typically, a collocation method is employed to solve for the Chebyshev coefficients. This means that the equations are required to be exactly satisfied at $N-1$ grid points defined by the equations (4.18) and (4.19). Excluded are the points $r = r_i$ and $r = r_o$, where the *boundary conditions* provide additional constraints on the set of Chebyshev coefficients.

4.5 Time-stepping schemes

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

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

$$\frac{\partial}{\partial t} y = \mathcal{I}(y,t) + \mathcal{E}(y,t), \quad y(t_o) = y_o \ . \tag{4.29}$$

where $\mathcal{I}$ denotes the terms treated in an implicit time step and $\mathcal{E}$ the terms treated explicitly, i.e. the nonlinear and Coriolis contributions. In MagIC, two families of time-stepping schemes are supported: IMEX multistep and IMEX multistage methods.

First of all, the IMEX multistep methods correpond to time schemes where the solution results from the combination of several previous steps (such as for instance the Crank-Nicolson/Adams-Bashforth scheme). In this case, a general k-step IMEX multistep scheme reads

$$\left(I - b_o^{\mathcal{I}} \delta t \, \mathcal{I}\right) y_{n+1} = \sum_{j=1}^{k} a_j y_{n+1-j} + \delta t \sum_{j=1}^{k} \left(b_j^{\mathcal{E}} \mathcal{E}_{n+1-j} + b_j^{\mathcal{I}} \mathcal{I}_{n+1-j}\right) \ ,$$

where I is the identity matrix. The vectors $\boldsymbol{a}$, $\boldsymbol{b}^{\mathcal{E}}$ and $\boldsymbol{b}^{\mathcal{I}}$ correspond to the weighting factors of an IMEX multistep scheme. For instance, the commonly-used second-order scheme assembled from the combination of a Crank-Nicolson for the implicit terms and a second-order Adams-Bashforth for the explicit terms (hereafter CNAB2) corresponds to the following vectors: $\boldsymbol{a} = (1, 0)$, $\boldsymbol{b}^{\mathcal{I}} = (1/2, 1/2)$ and $\boldsymbol{b}^{\mathcal{E}} = (3/2, -1/2)$ for a constant timestep size δt.

In addition to CNAB2, MagIC supports several semi-implicit backward differentiation schemes of second, third and fourth order that are known to have good stability properties (heareafter SBDF2, SBDF3 and SBDF4), a modified CNAB2 from Ascher et al. (1995) (termed MODCNAB) and the CNLF scheme (combination of Crank-Nicolson and Leap-Frog for the explicit terms).

MagIC also supports several IMEX Runge-Kutta multistage methods, frequently called DIRK, an acronym that stands for *Diagonally Implicit Runge Kutta*. For such schemes, the equation (4.29) is time-advanced from t_n to t_{n+1} by solving ν sub-stages

$$(I - a_{ii}^{\mathcal{I}}\delta t\mathcal{I})y_i = y_n + \delta t \sum_{j=1}^{i-1} \left(a_{i,j}^{\mathcal{E}}\mathcal{E}_j + a_{i,j}^{\mathcal{I}}\mathcal{I}_j\right), \quad 1 \leq i \leq \nu,$$

where y_i is the intermediate solution at the stage i. The matrices $a_{i,j}^{\mathcal{E}}$ and $a_{i,j}^{\mathcal{I}}$ constitute the so-called Butcher tables that correspond to a DIRK scheme. MagIC supports several second and third order schemes: ARS222 and ARS443 from Ascher et al. (1997), LZ232 from Liu and Zou (2006), PC2 from Jameson et al. (1981) and BPR353 from Boscarino et al. (2013).

In the code the equation (4.29) is formulated for each unknown spectral coefficient (expect pressure) of spherical harmonic degree ℓ and order m and for each radial grid point r_k. Because non-linear terms and the Coriolis force are treated explicitly, the equations decouple for all spherical harmonic modes. The different radial grid points, however, couple via the Chebychev modes and form a linear algebraic system of equations that can be solved with standard methods for the different spectral contributions.

For example the respective system of equations for the poloidal magnetic potential g time advanced with a CNAB2 reads

$$\left(\mathcal{A}_{kn} - \frac{1}{2}\,\delta t\,\mathcal{I}_{kn}\right)\, g_{\ell mn}(t+\delta t) = \left(\mathcal{A}_{kn} + \frac{1}{2}\,\delta t\,\mathcal{I}_{kn}\right)\, g_{\ell mn}(t) + \frac{3}{2}\,\delta t\,\mathcal{E}_{k\ell m}(t) - \frac{1}{2}\,\delta t\,\mathcal{E}_{k\ell m}(t-\delta t) \tag{4.30}$$

with

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

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

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

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

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

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

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

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

4.6 Coriolis force and nonlinear terms

4.6.1 Nonlinear terms entering the equation for W

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

$$\tilde{\rho}\left(\boldsymbol{u}\cdot\boldsymbol{\nabla}\boldsymbol{u}\right)=\begin{Bmatrix}\mathcal{A}_r\\ \mathcal{A}_\theta\\ \mathcal{A}_\phi\end{Bmatrix}=\begin{Bmatrix}-\tilde{\rho}\,E\left(u_r\dfrac{\partial u_r}{\partial r}+\dfrac{u_\theta}{r}\dfrac{\partial u_r}{\partial\theta}+\dfrac{u_\phi}{r\sin\theta}\dfrac{\partial u_r}{\partial\phi}-\dfrac{u_\theta^2+u_\phi^2}{r}\right)+\dfrac{1}{Pm}\left(j_\theta\,B_\phi-j_\phi\,B_\theta\right)\,,\\ -\tilde{\rho}\,E\left(u_r\dfrac{\partial u_\theta}{\partial r}+\dfrac{u_\theta}{r}\dfrac{\partial u_\theta}{\partial\theta}+\dfrac{u_\phi}{r\sin\theta}\dfrac{\partial u_\theta}{\partial\phi}+\dfrac{u_r u_\theta}{r}-\dfrac{\cos\theta}{r\sin\theta}u_\phi^2\right)+\dfrac{1}{Pm}\left(j_\phi\,B_r-j_r\,B_\phi\right)\,,\\ -\tilde{\rho}\,E\left(u_r\dfrac{\partial u_\phi}{\partial r}+\dfrac{u_\theta}{r}\dfrac{\partial u_\phi}{\partial\theta}+\dfrac{u_\phi}{r\sin\theta}\dfrac{\partial u_\phi}{\partial\phi}+\dfrac{u_r u_\phi}{r}+\dfrac{\cos\theta}{r\sin\theta}u_\theta u_\phi\right)+\dfrac{1}{Pm}\left(j_r\,B_\theta-j_\theta\,B_r\right)\,,\end{Bmatrix} \tag{4.31}$$

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

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

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

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

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

$$\boxed{\mathcal{N}^W_{\ell m}=\frac{2}{r}\left[im\frac{\partial W^m_\ell}{\partial r}-(\ell-1)c^m_\ell Z^m_{\ell-1}+(\ell+2)c^m_{\ell+1}Z^m_{\ell+1}\right]+{\mathcal{A}_r}^m_\ell\,.} \tag{4.32}$$

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 ${\mathcal{A}_r}^m_\ell$.

See also:

The final calculations of (4.32) are done in the subroutine *`get_td`*.

4.6.2 Nonlinear terms entering the equation for Z

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

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

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

$$\boldsymbol{e_r}\cdot\boldsymbol{\nabla}\times\left[(2\tilde{\rho}\boldsymbol{u})\times\boldsymbol{e_z}\right]=\frac{2}{r^2}\left(\vartheta_3\frac{\partial W}{\partial r}-\frac{1}{r}\vartheta_4 W+\frac{\partial Z}{\partial\phi}\right)\,. \tag{4.33}$$

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:

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

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

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

See also:

The final calculations of (4.35) are done in the subroutine `get_td`.

4.6.3 Nonlinear terms entering the equation for P

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

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

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

$$\boldsymbol{\nabla}_H\cdot[(2\tilde\rho\boldsymbol{u})\times\boldsymbol{e_z}]=\frac{2}{r^2}\left(-\frac{L_H}{r}\frac{\partial W}{\partial\phi}-\frac{\partial^2 W}{\partial\phi\partial r}+\vartheta_3\,Z\right). \tag{4.36}$$

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.

$$\vartheta_1\,\mathcal{A}t_\ell^m+\frac{\partial\mathcal{A}p_\ell^m}{\partial\phi}. \tag{4.37}$$

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

$$\begin{aligned}\mathcal{N}_{\ell m}^P=&\frac{2}{r^2}\left[-im\,\frac{\ell(\ell+1)}{r}\,W_\ell^m-im\,\frac{\partial W_\ell^m}{\partial r}+(\ell-1)(\ell+1)\,c_\ell^m\,Z_{\ell-1}^m+\ell(\ell+2)\,c_{\ell+1}^m\,Z_{\ell+1}^m\right]\\&+(\ell+1)\,c_\ell^m\,\mathcal{A}t_{\ell-1}^m-\ell\,c_{\ell+1}^m\,\mathcal{A}t_{\ell+1}^m+im\,\mathcal{A}p_\ell^m.\end{aligned} \tag{4.38}$$

See also:

The final calculations of (4.38) are done in the subroutine `get_td`.

4.6.4 Nonlinear terms entering the equation for s

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

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

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

Expanding this term leads to:

$$\begin{aligned}\mathcal{H}=&\frac{Pr\,Di}{Ra}\frac{1}{\tilde\rho\tilde T}\left[\tilde\rho\nu\left\{2\left(\frac{\partial u_r}{\partial r}\right)^2+2\left(\frac{1}{r}\frac{\partial u_\theta}{\partial\theta}+\frac{u_r}{r}\right)^2+2\left(\frac{1}{r\sin\theta}\frac{\partial u_\phi}{\partial\phi}+\frac{u_r}{r}+\frac{\cos\theta}{r\sin\theta}u_\theta\right)^2\right.\right.\\&+\left(r\frac{\partial}{\partial r}\left(\frac{u_\theta}{r}\right)+\frac{1}{r}\frac{\partial u_r}{\partial\theta}\right)^2+\left(r\frac{\partial}{\partial r}\left(\frac{u_\phi}{r}\right)+\frac{1}{r\sin\theta}\frac{\partial u_r}{\partial\phi}\right)^2\\&\left.\left.+\left(\frac{\sin\theta}{r}\frac{\partial}{\partial\theta}\left(\frac{u_\phi}{\sin\theta}\right)+\frac{1}{r\sin\theta}\frac{\partial u_\theta}{\partial\phi}\right)^2-\frac{2}{3}\left(\frac{d\ln\tilde\rho}{dr}\,u_r\right)^2\right\}\right.\\&\left.+\frac{\lambda}{Pm^2\,E}\left\{j_r^2+j_\theta^2+j_\phi^2\right\}\right].\end{aligned} \tag{4.39}$$

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

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

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`:

$$\mathcal{US}_r=\tilde\rho s u_r,\quad \mathcal{US}_\theta=\tilde\rho s u_\theta,\quad \mathcal{US}_\phi=\tilde\rho s u_\phi,$$

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

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

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

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

$$-\frac{1}{\tilde{\rho}}\left[\frac{1}{r^2}\frac{\partial\mathcal{USr}_\ell^m}{\partial r}+\vartheta_1\mathcal{USt}_\ell^m+\frac{\partial\mathcal{USp}_\ell^m}{\partial\phi}\right]. \tag{4.40}$$

Using (4.39) and (4.40), one thus finally gets

$$\boxed{\mathcal{N}_{\ell m}^S=-\frac{1}{\tilde{\rho}}\left[\frac{1}{r^2}\frac{\partial\mathcal{USr}_\ell^m}{\partial r}+(\ell+1)\,c_\ell^m\,\mathcal{USt}_{\ell-1}^m-\ell\,c_{\ell+1}^m\,\mathcal{USt}_{\ell+1}^m+im\mathcal{USp}_\ell^m\right]+\mathcal{H}_\ell^m.} \tag{4.41}$$

See also:

The θ and ϕ derivatives that enter (4.41) 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 ξ

The nonlinear term that enters the equation for chemical composition (4.25) is the advection term. This term is treated the same way as the advection term that enters the entropy equation. It is in a first step rearranged as follows

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

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

$$\mathcal{UX}_r=\tilde{\rho}\xi u_r,\quad \mathcal{US}_\theta=\tilde{\rho}\xi u_\theta,\quad \mathcal{UX}_\phi=\tilde{\rho}\xi u_\phi,$$

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

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

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 $r^2\mathcal{UX}_r$, $\mathcal{UX}_\phi/r\sin\theta$ and $\mathcal{UX}_\theta/r\sin\theta$ 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 $r^2\mathcal{UX}_r$, $\mathcal{UX}_\phi/r\sin\theta$ and $\mathcal{UX}_\theta/r\sin\theta$ 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 $\mathcal{UXr}_\ell^m$, $\mathcal{UXt}_\ell^m$ and $\mathcal{UXp}_\ell^m$ are defined.

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

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

One thus finally gets

$$\boxed{\mathcal{N}_{\ell m}^{\xi}=-\frac{1}{\tilde{\rho}}\left[\frac{1}{r^2}\frac{\partial \mathcal{UX}r_\ell^m}{\partial r}+(\ell+1)\,c_\ell^m\,\mathcal{UX}t_{\ell-1}^m-\ell\,c_{\ell+1}^m\,\mathcal{UX}t_{\ell+1}^m+im\,\mathcal{UX}p_\ell^m\right].} \tag{4.42}$$

See also:

The θ and ϕ derivatives that enter (4.42) are done in the subroutine `get_td`. The radial derivative is computed afterwards at the very beginning of `updateXi`.

4.6.6 Nonlinear terms entering the equation for g

The nonlinear term that enters the equation for the poloidal potential of the magnetic field (4.26) is the radial component of the induction term (4.28). In the following we introduce the electromotive force $\mathcal{F}=\boldsymbol{u}\times\boldsymbol{B}$ with its three components

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

The radial component of the induction term then reads:

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

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

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

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

We thus finally get

$$\boxed{\mathcal{N}_{\ell m}^{g}=(\ell+1)\,c_\ell^m\,{\mathcal{F}_\phi}_{\ell-1}^m-\ell\,c_{\ell+1}^m\,{\mathcal{F}_\phi}_{\ell+1}^m-im\,{\mathcal{F}_\theta}_\ell^m\,.} \tag{4.43}$$

See also:

The final calculations of (4.43) are done in the subroutine `get_td`.

4.6.7 Nonlinear terms entering the equation for h

The nonlinear term that enters the equation for the toroidal potential of the magnetic field (4.27) is the radial component of the curl of the induction term (4.28):

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

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

$$\frac{1}{r^2}\frac{\partial}{\partial r}\left[r^2\left(\vartheta_1\,\mathcal{F}t^m_\ell+\frac{\partial\,\mathcal{F}p^m_\ell}{\partial\phi}\right)\right]+L_H\,\mathcal{F}r^m_\ell\,.$$

We thus finally get

$$\boxed{\mathcal{N}^h_{\ell m}=\ell(\ell+1)\,\mathcal{F}r^m_\ell+\frac{1}{r^2}\frac{\partial}{\partial r}\left[r^2\left\{(\ell+1)\,c^m_\ell\,\mathcal{F}t^m_{\ell-1}-\ell\,c^m_{\ell+1}\,\mathcal{F}t^m_{\ell+1}+im\,\mathcal{F}p^m_\ell\right\}\right]\,.} \tag{4.44}$$

See also:

The θ and ϕ derivatives that enter (4.44) are computed in the subroutine `get_td`. The remaining radial derivative is computed afterwards at the very beginning of `updateB`.

4.7 Boundary conditions and inner core

4.7.1 Mechanical boundary conditions

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

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

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

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

at the respective boundary. In spectral space these conditions read

$$\mathcal{C}'_n(r)W_{\ell mn}=0\ \text{ at }\ r=r_i,r_o\,, \tag{4.46}$$

and

$$\mathcal{C}_n(r)Z_{\ell mn}=0\ \text{ at }\ r=r_i,r_o\,, \tag{4.47}$$

for all spherical harmonic modes (ℓ, m). The conditions (4.45)-(4.47) 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 (4.47) for the mode $(\ell = 1, m = 0)$:

$$\mathsf{I}\frac{\partial \omega}{\partial t} = \Gamma_L + \Gamma_\nu \, .$$

The tensor I denotes the moment of inertia of inner core or mantle, respectively, ω is the mantle or inner-core rotation rate relative to that of the reference frame, and $\Gamma_{L,\nu}$ are the respective torques associated with Lorentz or viscous forces. The torques are expressed by

$$\Gamma_L = \frac{1}{E\,Pm} \oint B_r B_\phi \, r \sin\theta \, \mathrm{d}S \, ,$$

and

$$\Gamma_\nu = \oint \tilde{\rho}\tilde{\nu} r \frac{\partial}{\partial r}\left(\frac{u_\phi}{r}\right) r \sin\theta \, \mathrm{d}S \, ,$$

where $\mathrm{d}S = r^2 \sin\theta \mathrm{d}\theta \mathrm{d}\phi$ and $r \in [r_i, r_o]$ in the above expressions. Using the following equality

$$\oint \tilde{\rho} r \sin\theta u_\phi \, \mathrm{d}S = 4\sqrt{\frac{\pi}{3}} Z_{10} r^2,$$

the viscous torques can be expressed by

$$\Gamma_\nu = \pm 4\sqrt{\frac{\pi}{3}} \tilde{\nu} r^2 \left[\frac{\partial Z_{10}}{\partial r} - \left(\frac{2}{r} + \beta\right) Z_{10}\right] \, ,$$

where the sign in front depends whether $r = r_o$ or $r = r_i$.

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

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

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

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

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

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

and

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

which can be fulfilled with

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

and

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

In spectral representation this then reads

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

4.7.2 Thermal boundary conditions

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

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

at r_i and/or r_o, while the latter means

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

In spectral representation for example the respective entropy condition read

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

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

4.7.3 Boundary conditions for chemical composition

For the chemical composition, either the value or the flux is imposed at the boundaries. The former implies:

$$\xi = \text{const.}$$

at r_i and/or r_o, while the latter means

$$\frac{\partial}{\partial r} \xi = \text{const.}$$

In spectral representation, this then reads

$$\mathcal{C}_n \xi_{\ell mn} = \text{const.} \quad \text{or} \quad \mathcal{C'}_n \xi_{\ell mn} = \text{const.}$$

4.7.4 Magnetic boundary conditions and inner core

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

$$\mathcal{C}_n(r) W_{\ell mn} = 0 \quad \text{at} \quad r = r_i, r_o \quad . \tag{4.48}$$

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

$$\mathcal{C}_n(r) g_{\ell mn} = 0 \quad , \quad \mathcal{C}'_n(r) g_{\ell mn} = 0 \quad \text{and} \quad \mathcal{C}_n(r) h_{\ell mn} = 0. \tag{4.49}$$

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

or below. We distinguish these contributions with upper indices I for internal or below and E for external or above. The total potential then has the form:

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

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

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

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

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

for the horizontal components and

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

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

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

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

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

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

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

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

$$\int d\Omega \, Y_\ell^{m\star} \, \boldsymbol{e_r} \cdot \left[\boldsymbol{\nabla} \times \left(\boldsymbol{u^I} \times \boldsymbol{B^I}\right)\right] = -i\,\omega_I\, m\, \frac{\ell(\ell+1)}{r^2}\, g^I_{\ell m}(r) \, , \tag{4.50}$$

$$\int d\Omega \, Y_\ell^{m\star} \, \boldsymbol{e_r} \cdot \left[\boldsymbol{\nabla} \times \boldsymbol{\nabla} \times \left(\boldsymbol{u^I} \times \boldsymbol{B^I}\right)\right] = -i\,\omega_I\, m\, \frac{\ell(\ell+1)}{r^2}\, h^I_{\ell m}(r) \, . \tag{4.51}$$

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

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

$$x(r) = \frac{r}{r_i} \quad , \quad -r_i \leq r \leq r_i \quad . \tag{4.52}$$

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

$$g^I_{\ell m}(r) = \left(\frac{r}{r_i}\right)^{\ell+1} \sum_{i=0}^{M-1} g^I_{\ell m\, 2i}\, \mathcal{C}_{2i}(r) \quad . \tag{4.53}$$

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 (4.26), (4.27) on the outer-core side and equations (4.50), (4.51) on the inner-core side. Employing free-slip conditions or allowing for electrical conductivity differences between inner and outer core leads to more complicated and even non-linear matching conditions.

CHAPTER

FIVE

CONTRIBUTING TO THE CODE

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

Contribute

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

5.1 Checking the consistency of the code

It is frequently required to check the consistency of the code, especially **after the implementation of new features**. For this reason, we developed the python test suite `magic_wizard.py`, 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.

You can run it as follows:

```
./magic_wizard.py <options>
```

It supports the following options:

```
-h,  --help              Show usage overview
     --level LEV               Run only tests from level LEV
     --use-debug-flags         Compile MagIC with the debug flags
     --use-mpi                 Use MPI
     --use-openmp              Use the hybrid version of MagIC
     --use-mkl                 Use the MKL for FFTs and Lapack calls
     --use-shtns               Use SHTns for Legendre transforms
     --use-precond USE_PRECOND Use matrix preconditioning
     --nranks NRANKS           Specify the number of MPI ranks
     --nthreads NTHREADS       Specify the number of threads (hybrid version)
     --mpicmd MPICMD           Specify the mpi executable (mpiexec, mpirun, srun)
```

Note: Make sure that your environment variables `FC` and `CC` are correctly defined otherwise the script will use the default system compilers.

The `--level LEV` option 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 (BM1) (Christensen et al., 2001) - start from zero (`dynamo_benchmark`) • Variable transport properties (viscosity, thermal diffusivity and electrical diffusivity) in an anelastic convective model (`varProps`) • Test of a case that uses finite differences - restart from a case that used Chebyshev polynomials (`finite_differences`) • Boussinesq dynamo benchmark (BM2) (Christensen et al., 2001) - start from a saturated state (`boussBenchSat`) • Double-diffusive convection benchmark (Breuer et al., 2010) - start from a saturated state (`doubleDiffusion`) • Axisymmetric spherical Couette flow - this auto-test checks the axisymmetric version of MagIC (`couetteAxi`) • Test Precession (`precession`) • Whole sphere benchmark (Marti et al., 2014) - start from a saturated state (`full_sphere`)
1	• Test reading and writing of restart files (`testRestart`) • Test different grid truncations (`testTruncations`) • Test mapping on to a new grid (`testMapping`) • Test different outputs produced (`testOutputs`) • Test different radial outputs - `*R.TAG` (`testRadialOutputs`)
2	• Hydrodynamic anelastic benchmark (Jones et al., 2011) (`hydro_bench_anel`)
3	• Heat flux perturbation (`fluxPerturbation`) • Isothermal model with $N_\rho = 3$ (`isothermal_nrho3`) • Boussinesq Dynamo benchmark for conducting and rotating inner core (`dynamo_benchmark_condICrotIC`) • Anelastic dynamo with variable conductivity (`varCond`)
4	• Test the writing of CMB and coeff files (`testCoeffOutputs`) • Test the writing of RMS force balance (`testRMSOutputs`) • Test the writing of Graphic and Movie files (`testGraphMovieOutputs`) • Test the writing of TO and Geos outputs (`testTOGeosOutputs`)

5.2 Advices when contributing to the code

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

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

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

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

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

5.3 Building the documentation and contributing to it

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

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

and build the html documentation

```
$ make html
```

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

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

```
$ make latexpdf
```

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

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

CHAPTER

SIX

INPUT PARAMETERS

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

An example for the short namelist defining inner core parameters is

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

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

Magic uses the following **eight namelists** :

Namelists

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

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

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

```
&grid
 n_r_max     =97,           ! 97 radial grid points
 n_cheb_max  =95,
 n_phi_tot   =288,          ! 288 points in the azimuthal direction
```

(continues on next page)

(continued from previous page)

```
 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",        ! Trailing name of the outputs produced by the code
 n_time_steps=50000,         ! Number of time steps
 courfac     =2.5D0,         ! Courant factor (flow)
 alffac      =1.0D0,         ! Courant factor (magnetic field)
 dtmax       =1.0D-4,        ! Maximum allowed time-step
 alpha       =0.6D0,
 runHours    =23,            ! Run time (hours)
 runMinutes  =30,            ! Run time (minutes)
 time_scheme ='CNAB2',       ! Name of the time stepper
/
&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  ="checkpoint_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 checkpoint_end.TAG file produced at the end of the run
 runid       ="C.Jones bench",
/
&mantle
 nRotMa      =0              ! Non-rotating mantle
/
&inner_core
 sigma_ratio =0.d0,          ! Non-conducting inner core
 nRotIC      =0,             ! Non-rotating inner core
/
```

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 ($[r_i, r_o]$). If Chebyshev polynomials are used for the radial integration scheme, there are some constraints on the value of *n_r_max*: first of all it must be of the form `4*n+1`, where `n` is an integer; and second the prime decomposition of *n_r_max-1* should only contain multiple of 3, 4 and 5 (this second condition comes from the limitations of the built-in discrete cosine transforms). This constraint is released when finite differences are used.

Note: If Chebyshev polynomials are used, the possible values for *n_r_max* below 220 are hence: 17, 21, 25, 33, 37, 41, 49, 61, 65, 73, 81, 97, 101, 109, 121, 129, 145, 161, 181, 193, 201, 217,...

- **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`. This quantity will not be used if finite differences are used.
- **n_phi_tot** (default *n_phi_tot=192*) is an integer which gives the number of longitudinal/azimuthal grid points. It has the following constraints:
  - *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, ...

- **l_axi** (default *l_axi=.false.*) is a logical. When set to true, one considers only the axisymmetric mode (i.e. MagIC becomes a 2-D axisymmetric code).
- **n_theta_axi** (default *n_theta_axi=0*) is an integer which gives the number of latitudinal grid points when MagIC computes only the axisymmetric mode.
- **fd_order** (default *fd_order=2*) is an integer. This is the order of the finite difference scheme for the bulk points (possible values are 2, 4, 6).
- **fd_order_bound** (default *fd_order_bound=2*) is an integer. This is the order of the finite difference scheme for the boundary points (possible values are 1,2,3,4,5,6, ...). This has to be smaller than the order of the scheme used for the bulk points.
- **fd_stretch** (default *fd_stretch=0.3*) is a real. It controls the ratio between the number of points in the boundary layers and in the bulk.
- **fd_ratio** (default *fd_ratio=0.1*) is a real. It controls the ratio between the smallest grid spacing and the largest grid spacing.

Note: When *fd_ratio* is set to 1, the radial grid is regularly-spaced.

- **l_var_l** (default *l_var_l=.false.*) is a logical. The spherical harmonic degree is a function of radius, when set to true. This practically reduces the number of spherical harmonic transforms in parts of the fluid domain but it comes at the price of an MPI inbalance. This feature is useful when computing full sphere geometry to avoid a too severe time step limitation close to the center. Right now the form of the radial dependence follows:

$$\ell(r) = 1 + (\ell_{\max} - 1)\sqrt{\frac{r}{r_o}}$$

6.1.2 Inner Core

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

6.1.3 Symmetry and aliasing

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

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

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

6.2 Control namelist

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

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

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

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

6.2.1 Default scales

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

n_tScale=0	Use viscous time scale.	d^2/ν
n_tScale=1	Use magnetic time scale.	d^2/η
n_tScale=2	Use thermal time scale.	d^2/κ
n_tScale=3	Use rotational time scale.	Ω^{-1}

- **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.
- **l_update_xi** (default *l_update_xi=.true.*) is a logical that specifies whether the chemical composition should be time-stepped or not.

6.2.3 Time step control

A modified Courant criterion 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. The parameter **l_cour_alf_damp** allows to choose whether the actual Alven speed is used to estimate the Courant condition or if damping is included. Practically, the timestep size is controlled as follows

$$\delta t < \min_V \left(c_I \, E, \, \frac{\delta r}{|u_r|}, \, \frac{\delta h}{u_h} \right)$$

where $u_h = (u_\theta^2 + u_\phi^2)^{1/2}$, $\delta h = \dfrac{r}{\sqrt{\ell(\ell+1)}}$, and δr is the radial grid interval. The first term in the left hand side accounts for the explicit treatment of the Coriolis term.

$$|u_r| = c_F|u_{F,r}| + c_A \frac{u_{A,r}^2}{\left[u_{A,r}^2 + \left(\frac{1+Pm^{-1}}{2\delta r}\right)^2\right]^{1/2}},$$

where $u_{F,r}$ is the radial component of the fluid velocity and $u_{A,r} = Br/\sqrt{E\,Pm}$ is the radial Alven velocity. The denominator of the rightmost term accounts for the damping of the Alven waves.

- **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. This parameter corresponds to c_F in the above equation.
- **alffac** (default `alffac=1.0`) is a real, used to scale Alfven-velocity in Courant criteria. This parameter corresponds to c_A in the above equation.
- **intfac** (default `intfac=0.15`) is a real, used to scale Coriolis factor in Courant criteria. This parameter corresponds to c_I in the above equation.
- **l_cour_alf_damp** (default `l_cour_alf_damp=.true.`) is a logical. This is used to decide whether the damping of the Alven waves is taken into account when estimating the Courant condition (see Christensen et al., GJI, 1999). At low Ekman numbers, this criterion might actually lead to spurious oscillations/instabilities of the code. When turn to False, $|u_r| = c_F|u_{F,r}| + c_A|u_{A,r}|$.
- **time_scheme** (default `time_scheme='CNAB2'`) is a character string. This is used to choose the time step integrator used in the code among the following implicit-explicit time schemes:

time_scheme='CNAB2'	Crank-Nicolson and 2nd order Adams-Bashforth scheme
time_scheme='CNLF'	Crank-Nicolson and Leap-Frog scheme
time_scheme='MODCNAB'	Modified CN/AB2
time_scheme='SBDF2'	Semi-implicit backward difference scheme of 2nd order
time_scheme='SBDF3'	Semi-implicit backward difference scheme of 3rd order
time_scheme='SBDF4'	Semi-implicit backward difference scheme of 4th order
time_scheme='ARS222'	Semi-implicit S-DIRK of 2nd order
time_scheme='ARS232'	Semi-implicit S-DIRK of 2nd order
time_scheme='CK232'	Semi-implicit S-DIRK of 2nd order
time_scheme='LZ232'	Semi-implicit S-DIRK of 2nd order
time_scheme='CB3'	Semi-implicit S-DIRK of 3rd order
time_scheme='ARS343'	Semi-implicit S-DIRK of 3rd order
time_scheme='ARS443'	Semi-implicit S-DIRK of 3rd order
time_scheme='BPR353'	Semi-implicit S-DIRK of 3rd order
time_scheme='LZ453'	Semi-implicit S-DIRK of 3rd order
time_scheme='KC343'	Semi-implicit S-DIRK of 3rd order
time_scheme='KC564'	Semi-implicit S-DIRK of 4th order
time_scheme='KC674'	Semi-implicit S-DIRK of 4th order
time_scheme='KC785'	Semi-implicit S-DIRK of 5th order

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.
- **difchem** (default *difchem=0.0*) is a real. This is the amplitude D of the hyperdiffusion applied to chemical composition.
- **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 Radial scheme and mapping of the Gauss-Lobatto grid

In MagIC, one can either use finite differences or Chebyshev polynomials for the radial integration scheme. This choice is controlled by the following input parameter:

- **radial_scheme** (default *radial_scheme='CHEB'*) is a character string.

radial_scheme='CHEB'	Use Chebyshev polynomials
radial_scheme='FD'	Use finite differences

When Chebyshev polynomials are used, it is also possible to use a non-linear mapping function to concentrate/diperse grid points around a point inside the domain.

- **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*.
- **map_function** (default *map_function='arcsin'*) is a character string. This allows to select which mapping function is used:

map_function='TAN'	Use a tangent mapping (see Bayliss and Turkel 1992)
map_function='ARCSIN'	Use finite differences (see Kosloff and Tal-Ezer 1993)

If the tangent mapping is used, the function that re-distributes the collocation points is expressed by

$$r = \frac{1}{2}\left(\alpha_2 + \frac{\tan\left[\lambda(r_{cheb} - x_0)\right]}{\alpha_1}\right) + \frac{r_i + r_o}{2} ,$$

where the Gauss-Lobatto collocation points are

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

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

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

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

If the arcsin mapping is used, the function that re-distributes the collocation points is given by

$$r = \frac{1}{2}\left[\frac{\arcsin\left(\alpha_1 r_{cheb}\right)}{\arcsin\alpha_1}\right] + \frac{r_i + r_o}{2} ,$$

In the Kosloff and Tal-Ezer mapping, α_1 transforms the Gauss-Lobatto grid into a more regularly-spaced grid. When $\alpha_1 \to 0$ one recovers the Gauss-Lobatto grid, while $\alpha_1 \to 1$ yields a regular grid.

Warning: The Kosloff-Tal-Ezer mapping becomes singular when $\alpha_1 = 1$. Acceptable values are $0 < \alpha_1 < 1$. Note that the error increases as $\epsilon = \left(\frac{1-\sqrt{1-\alpha_1^2}}{\alpha_1}\right)^{N_r}$.

- **alph1** (default `alph1=0.8`) 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.
- **anelastic_flavour** (default `anelastic_flavour="None"`) is a character string. This allows to change the thermal diffusion operator used within the anelastic approximation. Possible values are:

anelastic_flavour='LBR'	Entropy diffusion
anelastic_flavour='ENT'	Entropy diffusion
anelastic_flavour='ALA'	Anelastic liquid approximation
anelastic_flavour='TDIFF'	Temperature diffusion
anelastic_flavour='TEMP'	Temperature diffusion

- **polo_flow_eq** (default `polo_flow_eq="WP"`) is a character string. This allows to change how the equation for the poloidal flow potential is constructed. One can either use the radial component of the Navier-Stokes equation and hence keep a coupled system that involve the poloidal potential W and the pressure p, or take the radial component of the double-curl of the Navier-Stokes equation to suppress pressure.

polo_flow_eq='WP'	Use the pressure formulation
polo_flow_eq='DC'	Use the double-curl formulation

- **mpi_transp** (default `mpi_transp="auto"`) is a character string. It allows to change the way the global MPI transposes are handled by the code. By default, the code tries to determine by itself the fastest method. One can nevertheless force the code to use local communicators (such as Isend/Irecv/waitall), make use of the native alltoallv MPI variant or choose the alltoallw variant instead.

mpi_transp='auto'	Automatic determination of the fastest transpose
mpi_transp='p2p'	Use Isend/Irecv/Waitall communicators
mpi_transp='a2av'	Use alltoallv communicators
mpi_transp='a2aw'	Use alltoallw communicators

- **mpi_packing** (default `mpi_packing="packed"`) is a character string. It allows to change the size of the global MPI transposes. One can choose between some packing of the fields into buffers (default) or a sequence of single field transposes. There is a possible automatic detection but testing unfortunately reveals frequent false detection.

mpi_packing='auto'	Automatic determination of the fastest transpose
mpi_packing='packed'	Pack some fields into buffers
mpi_packing='single'	Transpose each field individually

- **l_adv_curl** (default `l_adv_curl=.false.`) is a logical. When set to True, the advection term is treated as $\boldsymbol{u} \times \boldsymbol{\omega}$ instead of $\boldsymbol{u}\boldsymbol{\nabla}\boldsymbol{u}$. The practical consequence of that is to reduce the number of spectral/spatial Spherical Harmonic Transforms and hence to speed-up the code. Because of the treatment of the viscous heating term in the anelastic approximation, this is only an option when considering Boussinesq models.

6.3 Physical parameters namelist

This namelist contains all the appropriate relevant control physical parameters.

6.3.1 Dimensionless control parameters

- **ra** (default `ra=0.0`) is a real. This the thermal Rayleigh number expressed by

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

- **raxi** (default `raxi=0.0`) is a real. This the compositional Rayleigh number expressed by

$$Ra_\xi = \frac{\alpha g_o \Delta \xi d^3}{\kappa_\xi \nu}$$

- **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

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

- **sc** (default `sc=10.0`) is a real. This is the Schmidt number expressed by

$$Sc = \frac{\nu}{\kappa_\xi}$$

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

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

- **po** (default `po=0.0`) is a real. This is the Poincaré number expressed by

$$Po = \frac{\Omega_p}{\Omega}$$

- **prec_angle** (default `prec_angle=23.5`) is a real. This is the angle between the precession and the rotation axes expressed in degrees.
- **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:

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

- **DissNb** (default `DissNb=0.0`) is a real. This is the dissipation number:

$$Di = \frac{\alpha_o g_o d}{c_p}$$

> **Warning:** This can only be provided as a **replacement** input of **strat**. I.E., when one wants to define a reference state, one has to specify **either** `strat` **or** `DissNb` in the input namelist.

- **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.

- **epscxi0** (default `epscxi0=0.0`) is a real. This is the volumetric source ϵ_ξ that enters the compositional equilibrium relation:

$$-\nabla \cdot (\tilde{\rho} \nabla \xi) + \epsilon_\xi = 0 \tag{6.2}$$

- **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. $f(r) = \tilde{\rho}(r)$.
`nVarEps=2`	Proportional to density times temperature, i.e. $f(r) = \tilde{\rho}(r)\tilde{T}$.

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

$$g(r) = g_0 + g_1 \frac{r}{r_o} + g_2 \left(\frac{r_o}{r}\right)^2 \tag{6.3}$$

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 (6.3).
- **g1** (default *g1=1*) is the pre-factor of the linear part of the gravity profile, i.e. g_1 in equation (6.3).
- **g2** (default *g2=0*) is the pre-factor of the $1/r^2$ part of the gravity profile, i.e. g_2 in equation (6.3).

6.3.5 Centrifugal acceleration

The centrifugal acceleration can be computed for a polytropic background

- **dilution_fac** (default *dilution_fac=0.0*) is the ratio of the centrifugal acceleration at the equator to the surface gravitational acceleration.

$$m = \frac{\Omega^2 d}{g_o} \tag{6.4}$$

6.3.6 Phase field

- **stef** (default *stef=1.0* is a real. This is the Stefan number (used when the phase field is plugged in). It is expressed by the ratio of the latent heat per unit mass associated with the solid-liquid transition and the specific heat:

$$St = \frac{\mathcal{L}}{c_p \Delta T}$$

- **tmelt** (default *tmelt=0.0*) is a real. This is the dimensionless melting temperature.
- **epsPhase** (default `epsPhase=0.01`) is a real. This is the dimensionless interface thickness between the solid and the liquid phase (sometimes known as the Cahn number).
- **phaseDiffFac** (default `phaseDiffFac=1.0`) is a real. This is a coefficient that goes in front of the diffusion term in the phase field equation.
- **penaltyFac** (default `penaltyFac=1.0`) is a real. This is coefficient used for the penalisation of the velocity field in the solid phase. The smaller the coefficient, the stronger the penalisation. Since this is a nonlinear term, it is handled explicitly and the time step size should be decreased with the square of `penaltyFac`.

6.3.7 Transport properties

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

Electrical conductivity

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

$$\begin{aligned} \sigma(r) &= 1 + (\sigma_m - 1)\left(\frac{r - r_i}{r_m - r_i}\right)^a \quad \text{for} \quad r < r_m, \\ \sigma(r) &= \sigma_m \exp\left[a\left(\frac{r - r_m}{r_m - r_i}\right)\frac{\sigma_m - 1}{\sigma_m}\right] \quad \text{for} \quad r \geq r_m. \end{aligned} \tag{6.5}$$

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

`nVarCond=0`	Constant electrical conductivity, i.e. $\sigma = \text{cst}$.
`nVarCond=1`	$\sigma \propto \tanh[a(r - r_m)]$
`nVarCond=2`	See equation (6.5).
`nVarCond=3`	**Magnetic diffusivity proportional to $1/\tilde{\rho}$, i.e.** $\lambda = \frac{\tilde{\rho}_i}{\tilde{\rho}}$
`nVarCond=4`	**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 (6.5).
- **con_DecRate** (default *con_DecRate=9*) is an integer. This defines the decay rate a that enters equation (6.5).
- **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 (6.5).
- **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`.

6.3.8 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.

6.3.9 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 temperature (Boussinesq) or entropy (anelastic) at outer boundary: $s(r_o) = s_{top}$
`ktops=2`	Fixed temperature gradient (Boussinesq) or entropy gradient at outer boundary: $\partial s(r_o)/\partial r = q_t$
`ktops=3`	Only use it in anelastic models: fixed temperature at outer boundary: $T(r_o) = T_{top}$
`ktops=4`	Only use it in anelastic models: fixed temperature gradient at outer boundary: $\partial T(r_o)/\partial r = q_t$

- **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)*.

Boundary conditions for chemical composition

- **ktopxi** (default *ktopxi=1*) is an integer to specify the outer boundary chemical composition boundary condition:

ktopxi=1	Fixed composition at outer boundary: $\xi(r_o) = \xi_{top}$
ktopxi=2	Fixed composition gradient at outer boundary: $\partial\xi(r_o)/\partial r = q_t$

- **kbotxi** (default *ktopxi=1*) is an integer to specify the inner boundary chemical composition boundary condition.
- **xi_top** (default *xi_top= 0 0 0.0 0.0*) is a real array of lateraly varying outer chemical composition 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:

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

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

Boundary conditions for phase field

- **ktopphi** (default *ktopphi=1*) is an integer to specify the boundary condition of the phase field at the outer boundary.

ktopphi=1	Fixed phase field at outer boundary: $\phi(r_o) = \phi_{top}$
ktopphi=2	Fixed phase field gradient : $\partial\phi(r_o)/\partial r = 0$

- **kbotphi** (default *kbotphi=1*) is an integer to specify the boundary condition of the phase field at the inner boundary.

Mechanical boundary conditions

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

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

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

Magnetic boundary conditions

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

ktopb=1	**Insulating outer boundary:** $$\frac{\partial g_{\ell m}}{\partial r} + \frac{\ell}{r} g_{\ell m} = 0, \quad \frac{\partial h_{\ell m}}{\partial r} = 0$$
ktopb=2	**Perfect condutor:** $$g_{\ell m} = \frac{\partial^2 g_{\ell m}}{\partial r^2} = 0, \quad \frac{\partial h_{\ell m}}{\partial r} = 0$$
ktopb=3	Finitely conducting mantle
ktopb=4	**Pseudo-vacuum outer boundary:** $$\frac{\partial g_{\ell m}}{\partial r} = 0, \quad h_{\ell m} = 0$$

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

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

Boundary condition for spherically-symmetric pressure

- **ktopp** (default *ktopp=1*) is an integer, which corresponds to the boundary condition for the spherically-symmetric pressure at $r = r_o$.

ktopp=1	The integral of the spherically-symmetric density perturbation vanishes.
ktopp=2	The spherically-symmetric pressure fluctuation vanishes at the outer boundary.

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 *checkpoint_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_xi** (default *scale_xi=1.0*) is a real. It can be possibly used to multiply the input chemical composition field from start_file by a constant factor scale_xi.

- **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 chemical composition

The chemical composition equation with possible volumetric sources and sinks given by *epscxi0* 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:

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

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_xi1` or `amp_xi2` at the mid-shell radius $r_i + 1/2$.

- **init_xi1** (default *init_xi1=0*) is an integer that controls the initial chemical composition. It follows the same specifications as `init_s1`.
- **amp_xi1** (default *amp_xi1=0.0*) is a real used to contol the amplitude of the perturbation defined by `init_xi1`.
- **init_xi2** (default *init_xi2=0*) is an integer that controls a second spherical harmonic degee. It follows the same specifications as `init_s1`.
- **amp_xi2** (default *amp_xi2=0.0*) is a real used to contol the amplitude of the perturbation defined by `init_xi2`.

Initialisation of phase field

- **init_phi** (default *init_phi=0*) is a integer used to specify the initial phase field. If `init_phi /= 0` a tanh profile centered around the melting temperature is used.

Initialisation of magnetic field

- **init_b1** (default *init_b1=0*) is an integer that controls the initial magnetic field. The following values are possible:
  - `init_b1<0`: random noise initialization of all (ℓ, m) modes, except for $(\ell = 0, m = 0)$. The subroutine `initB` in the file `init_fields.f90` contains the details of the implementation.
  - `init_b1=0`: nothing is initialized
  - `init_b1=1`: diffusive toroidal field initialized. Mode determined by `imagcon`.
  - `init_b1=2`: $(\ell = 1, m = 0)$ toroidal field with a maximum field strength of `amp_b1`. The radial dependence is defined, such that the field vanishes at both the inner and outer boundaries. In case of an insulating inner core: $h(r) \approx r\,\sin[\phi(r - r_o)]$. In case of a conducting inner core: $h(r) \approx r\,\sin[\pi(r/r_o)]$.
  - `init_b1=3`: $(\ell = 1, m = 0)$ poloidal field whose field strength is `amp_b1` at $r = r_i$. The radial dependence is chosen such that the current density j is independent of r:, i.e. $\partial j/\partial r = 0$. $(\ell = 2, m = 0)$ toroidal field with maximum strength `amp_b1`.
  - `init_b1=4`: $(\ell = 1, m = 0)$ poloidal field as if the core were an insulator (potential field). Field strength at $r = r_i$ is again given by `amp_b1`.

  - `init_b1=5`: $(\ell = 1, m = 0)$ poloidal field with field strength `amp_b1` at $r = r_i$. The radial dependence is again defined by $\partial j/\partial r = 0$.
  - `init_b1=6`: $(\ell = 1, m = 0)$ poloidal field independend of r.
  - `init_b1=7`: $(\ell = 1, m = 0)$ poloidal field which fulfills symmetry condition in inner core: $g(r) \approx \left(\frac{r}{r_i}\right)^2 \left[1 - \frac{3}{5}\left(\frac{r}{r_o}\right)^2\right]$. The field strength is given by `amp_b1` at $r = r_o$.
  - `init_b1=8`: same poloidal field as for `init_b1=7`. The toroidal field fulfills symmetry conditions in inner core and has a field strength of `amp_b1` at $r = r_i$: $h(r) \approx \left(\frac{r}{r_i}\right)^3 \left[1 - \left(\frac{r}{r_o}\right)^2\right]$.
  - `init_b1=9`: $(\ell = 2, m = 0)$ poloidal field, which is a potential field at the outer boundary.
  - `init_b1=10`: equatorial dipole only.
  - `init_b1=11`: axial and equatorial dipoles.
  - `init_b1=21`: toroidal field created by inner core rotation, equatorially symmetric $(\ell = 1, m = 0)$: $h(r) =$ ampb1 $\left(\frac{r_i}{r}\right)^6$. The field strength is given by `amp_b1` at $r = r_i$.
  - `init_b1=22`: toroidal field created by inner core rotation, equatorially antisymmetric $(\ell = 2, m = 0)$. Same radial function as for `init_b1=21`.

- **amp_b1** (default *amp_b1=0.0*) is a real used to contol the amplitude of the function defined by `init_b1`.
- **imagcon** (default *imagcon=0*) is an integer, which determines the imposed magnetic field for magnetoconvection. The magnetic field is imposed at boundaries.
  - `imagcon=0`: no magneto-convection
  - `imagcon<0`: axial poloidal dipole imposed at ICB with a maximum magnetic field strength `amp_b1`.
  - `imagcon=10`: $(\ell = 2, m = 0)$ toroidal field imposed at ICB and CMB with a maximum amplitude `amp_b1` at both boundaries.
  - `imagcon=11`: same as `imagcon=10` but the maximum amplitude is now `amp_b1` at the ICB and `-amp_b1` at the CMB.
  - `imagcon=12`: $(\ell = 1, m = 0)$ toroidal field with a maximum amplitude of `amp_b1` at the ICB and the CMB.
- **tmagcon** (*tmagcon=0.0*) is a real.

Initialisation of velocity field

- **init_v1** (default *init_v1=0*) is an integer that controls the initial velocity. The following values are possible:
  - `init_v1=0`: nothing is initialized
  - `init_v1=1`: a differential rotation profile of the form

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

    where $s = r\sin\theta$ is the cylindrical radius. This profile only makes sense when one studies spherical Couette flows.
  - `init_v1=2`: a differential rotation profile of the form $\Omega = \frac{\text{ampv1}}{\sqrt{1+s^4}}$ is introduced.
  - `init_v1>2`: a random-noise of amplitude `amp_v1` is initialised. The subroutine `initV` in `init_fields.f90` gives the detail of this implementation.

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

6.6 Output control namelist

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

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

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

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

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

---

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

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

---

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

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

---

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

Note: These input variables are usually named with a pattern that follows `t_outputName`, for instance, *t_graph*, *t_rst*, *t_spec*, *t_log*, *t_movie*, etc.

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

Possible outputs

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 (*heat.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 (*checkpoint_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.
- **l_2D_spectra** (default `l_2D_spectra=.false.`) is a logical. When set to `.true.`, this logical enables the calculation of 2-D spectra in the (r, ℓ) and in the (r, m) parameter spaces. Those data are stored in the files named *2D_[mag|kin]_spec_#.TAG*.

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 $\partial s/\partial r$
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.

continues on next page

Table 1 – continued from previous page

Keyword	Fields stored in movie file
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.

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

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 radial space

The **[V|B|T]_lmr** outputs controls the output of potential files (*V_lmr_#.TAG*, *B_lmr_#.TAG* and *T_lmr_#.TAG*). These are files contain the poloidal and toroidal flow and magnetic field potentials (and entropy/temperature) written in spectral and radial spaces (for instance `w(lm_max, n_r_max)`). These files can be quite handy since they can be possibly used to reconstruct any quantity in the spectral space or in the physical space you may be interested in.

Standard inputs

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

- **t_TOmovie** (default *t_TOmovie=-1.0 -1.0 ...*) is real array, which contains the times when TO_mov outputs are requested.
- **dt_TOmovie** (default dt_TOmovie=0.0) is a real, which defines the time interval between TO_mov outputs.
- **t_TOmovie_start** (default *t_TOmovie_start=0.0*) is a real, which defines the time to start writing TO_mov outputs.
- **t_TOmovie_stop** (default *t_TOmovie_stop=0.0*) is a real, which defines the time to stop writing TO_mov outputs.

6.6.11 RMS force balance

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

- **l_RMS** (default *l_RMS=.false.*) is a logical, which enables the calculation of RMS force balance, when set to .true.. The outputs are stored in *dtVrms.TAG*, *dtBrms.TAG* and *dtVrms_spec.TAG*.
- **rCut** (default *rCut=0.0*) is a float. This is the thickness of the layer which is left out at both boundaries for the RMS calculation. rCut=0.075 actually means that 7.5% below the CMB and above the ICB are disregarded in the force balance calculation.
- **rDea** (default *rDea=0.0*) is a float. This controls the dealiasing in RMS calculations. rDea=0.1 means that the highest 10% of the Chebyshev modes are set to zero.
- **l_2D_RMS** (default *l_2D_RMS=.false.*) is a logical. When set to .true., this logical enables the calculation of 2-D force balance in the (r, ℓ) and parameter space. Those data are stored in the files named *2D_dtVrms_spec.TAG*.

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 calculations can be found in the subroutine getEgeos in the Egeos.f90 file. These quantities are then stored in the columns 10-16 of the *geos.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 *helicity.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*.

Earth-likeness of the CMB field

- **l_earth_likeness** (default *l_earth_likeness=.false.*) is a logical. When set to `.true.`, this logical enables the calculation of the Earth-likeness of the CMB magnetic field following (Christensen et al., 2010). The time series of the four criteria are stored in *earth_like.TAG*.
- **l_max_comp** (default *l_max_comp=8*) is an integer. This is the maximum spherical harmonic degree used to calculate the Earth-likeness of the CMB field.
- **l_geo** (default *l_geo=11*) is an integer. This is the maximum spherical harmonic degree used to compute the dipolarity of the magnetic field at the CMB. This is used to compute columns 6 and 15 of the *dipole.TAG* file.

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*.

Pressure

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

Time evolution of the m-spectra

- **l_energy_modes** (default *l_energy_modes=.false.*) is a logical. When set to `.true.`, this logical enables the storage of the time-evolution of the kinetic and magnetic energy spectra for a given range of spherical harmonic orders: *time spectra*.
- **m_max_modes** (default *m_max_modes=13*) is an integer. This controls the maximum spherical harmonic order when *l_energy_modes=.true.*.

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=-1).
- **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))
```

The following defines the parameters when one wants to excite inertial modes in the system artificially using a method similar to Rieutord et. al 2012 .

- **amp_RiMaSym** (default amp_RiMaSym=0.0) is a real which defines the amplitude of forcing on the outer boundary for an equatorially symmetric mode
- **omega_RiMaSym** (default omega_RiMaSym=0.0) is a real which defines the frequency of forcing on the outer boundary for an equatorially symmetric mode
- **m_RiMaSym** (default m_RiMaSym=0.0) is an integer which defines the wavenumber of the equatorially symmetric mode one wants to excite

The following variables define the same for an equatorially anti-symmetric mode:

- **amp_RiMaAsym** (default amp_RiMaAsym=0.0)
- **omega_RiMaAsym** (default omega_RiMaAsym=0.0)
- **m_RiMaAsym** (default m_RiMaAsym=0.0)

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))
```

The following, as for the mantle namelist, is for artificially exciting inertial modes in the spherical shell, but for the inner boundary.

- **amp_RiIcSym** (default `amp_RiIcSym=0.0`) is a real which defines the amplitude of forcing on the inner boundary for an equatorially symmetric mode
- **omega_RiIcSym** (default `omega_RiIcSym=0.0`) is a real which defines the frequency of forcing on the inner boundary for an equatorially symmetric mode
- **m_RiIcSym** (default `m_RiIcSym=0.0`) is an integer which defines the wavenumber of the equatorially symmetric mode one wants to excite

The following variables define the same for an equatorially anti-symmetric mode:

- **amp_RiIcAsym** (default `amp_RiIcAsym=0.0`)
- **omega_RiIcAsym** (default `omega_RiIcAsym=0.0`)
- **m_RiIcAsym** (default `m_RiIcAsym=0.0`)

CHAPTER

SEVEN

INTERACTIVE COMMUNICATION WITH THE CODE USING `SIGNAL.TAG`

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

- **END**: Changing the word `NOT` to `END` will cause the code to finish after the current time step and write all the outputs as if it was programmed to finish at that time from the start. This will thus normally produce the *checkpoint_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 *checkpoint_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`.
- **POT**: Changing the word `NOT` to `POT` will cause the code to produce the potential files *V_lmr_#.TAG* (and possibly *B_lmr_#.TAG*, *T_lmr_#.TAG* <*secPotFiles*> and *Xi_lmr_#.TAG* <*secPotFiles*> depending if the run is magnetic or not, or if it solves a temperature/entropy and/or chemical composition equations). Once the potential files have been written, the keyword will be automatically replaced by `NOT`.

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

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

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

CHAPTER

EIGHT

OUTPUT FILES

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

1. Most of the information found in `STDOUT` is also written to the **log-file** called *log.TAG*. In addition, this file contains all input parameters, truncation, information on other output files, and some results like the time averaged energies (when *l_average=.true.*).
2. There are several ascii files that contain the **time-evolution of integrated quantities** (energies, heat fluxes, rotation rate, Reynolds numbers, etc.) that are systematically produced:
   - Kinetic energies: *e_kin.TAG*,
   - Magnetic energies: *e_mag_oc.TAG* and *e_mag_ic.TAG*,
   - Rotation rates: *rot.TAG*,
   - Informations about the dipolar component of the magnetic field: *dipole.TAG*,
   - Diagnostic parameters (Reynolds, Elsasser, etc.): *par.TAG*,
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*,
   - Heat transport: *heat.TAG*,
   - Helicity: *helicity.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*,
   - Geostrophy: *geos.TAG*,
   - RMS calculations of the force balances: *dtVrms.TAG* and *dtBrms.TAG*,
   - Kinetic energies perpendicular and parallel to the rotation axis: *perpPar.TAG*.
4. **Time-averaged radial profiles**:
   - Kinetic energies: *eKinR.TAG*,
   - Magnetic energies: *eMagR.TAG*,
   - Diagnostic quantities: *parR.TAG*,

- Power budget: *powerR.TAG*,
- Average temperature, entropy and pressure: *heatR.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]_spec_#.TAG*.
   - Time-averaged 2-D ([r,ell] and [r,m]) spectra: *2D_[mag|kin]_spec_ave.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**: *checkpoint_end.TAG*.
10. **Time-evolution of the poloidal and toroidal coefficients** at different 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: *V_lmr_#.TAG*, *B_lmr_#.TAG* and *T_lmr_#.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*, *checkpoint_#.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_spec_avg=.true.*, this section also provides information about average spectra being written. If *l_average=.true.*, it is additionally mentioned that time averaged graphic files are written.
- **Wall times**: this is the last part of the log file and it provides information about the mean wall time for running different parts of the code. These values can be used to judge the speed and scaling capabilities of your computer.

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

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

8.2 Default time-series outputs

8.2.1 e_kin.TAG

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

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

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

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

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

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

8.2.2 e_mag_oc.TAG

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

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

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

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

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

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

8.2.3 e_mag_ic.TAG

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

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

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

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

8.2.4 rot.TAG

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

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

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

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

8.2.5 dipole.TAG

This file contains several informations about the magnetic dipole. This file is written by the subroutine *get_e_mag*. The maximum degree used to compute columns 6 and 15 is given by *l_geo*.

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 dmB
16	Elsasser number at CMB
17	Local Rol based on non-ax. flow
18	Convective flow length scale dlVc
19	Peak of the poloidal kinetic energy
20	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)
```

8.3 Additional optional time-series outputs

8.3.1 heat.TAG

This files contains informations about the heat transfer (Nusselt number, entropy and temperature at both boundaries). This file is written by the subroutine *outHeat*.

No. of column	Contents
1	time
2	Nusselt number at the inner boundary
3	Nusselt number at the outer boundary
4	Nusselt number based on ΔT ratio
5	Temperature at the inner boundary
6	Temperature at the outer boundary
7	Entropy at the inner boundary
8	Entropy at the outer boundary
9	Heat flux at the inner boundary
10	Heat flux at the outer boundary
11	Pressure perturbation at the outer boundary
12	volume integrated mass perturbation
13	Sherwood number at the inner boundary
14	Sherwood number at the outer boundary
15	Sherwood number based on $\Delta \xi$ ratio
16	Chemical composition at the inner boundary
17	Chemical composition at the outer boundary

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

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

8.3.2 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)
```

8.3.3 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	Chemical power: $Ra_\xi\, g(r)\, \langle u_r \xi' \rangle_s$
4	Viscous power at the inner boundary (ICB)
5	Viscous power at the outer boundary (CMB)
6	Viscous dissipation: $\langle (\nabla \times u)^2 \rangle_s$
7	Ohmic dissipation: $\langle (\nabla \times B)^2 \rangle_s$
8	Total power at the CMB (viscous + Lorentz)
9	Total power at the ICB (viscous + Lorentz)
10	Total power
11	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*')
```

8.3.4 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

8.3.5 earth_like.TAG

This contains informations about the Earth-likeness of the CMB radial magnetic field. This file is written by the subroutine *get_e_mag*.

Note: This file is **only** calculated when *l_earth_like=.true.*.

No. of column	Contents
1	time
2	Ratio of axial dipole to non-dipole component at the CMB
3	Equatorial symmetry of the CMB field (odd/even ratio)
4	Zonality: zonal to non-zonal ratio of the CMB field
5	Magnetic flux concentration at the CMB

The details of the calculations are given in (Christensen et al., 2010).

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

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

8.3.6 geos.TAG

This file contains informations about the geostrophy of the flow. This file is written by the subroutine `getEgeos`.

Note: This file is **only** calculated when *l_par=.true.*.

No. of column	Contents
1	time
2	Relative geostrophic kinetic energy
3	Relative kinetic energy in the northern part of the TC
4	Relative kinetic energy in the southern part of the TC
5	Kinetic energy (calculated on the cylindrical grid)
6	North/South correlation of Vz, outside the TC
7	North/South correlation of vorticity outside the TC
8	North/South correlation of helicity outside the TC
9	Geostrophy of axisymmetic flow
10	Geostrophy of zonal flow
11	Geostrophy of meridional flow
12	Geostrophy of non-axisymmetric flow

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

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

8.3.7 helicity.TAG

This files contains informations about the kinetic helicity in both the Northern and the Southern hemispheres. This file is written by the subroutine *outHelicity*.

Note: This file is **only** calculated when *l_hel=.true.*.

No. of column	Contents
1	time
2	Helicity (northern hemisphere)
3	Helicity (southern hemisphere)
4	RMS helicity (northern hemisphere)
5	RMS helicity (southern hemisphere)
6	Helicity (northern hemisphere, only non-axisym. flow)
6	Helicity (southern hemisphere, only non-axisym. flow)
8	RMS helicity (northern hemisphere, only non-axisym. flow)
9	RMS helicity (southern hemisphere, only non-axisym. flow)

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

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

8.3.8 u_square.TAG

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

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

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

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

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

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

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

8.3.9 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.

8.3.10 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.

8.3.11 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

8.3.12 dtVrms.TAG

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

This files contains the RMS force balance of the Navier Stokes equation. This file is written by the subroutine *dtVrms*.

No. of column	Contents
1	Time
2	Total inertia: dU/dt and advection
3	Coriolis force
4	Lorentz force
5	Advection term
6	Diffusion term
7	Thermal buoyancy term
8	Chemical buoyancy term
9	Pressure gradient term
10	Sum of force terms: geostrophic balance
11	Sum of force terms: pressure, Coriolis and Lorentz
12	Sum of force terms: pressure, buoyancy and Coriolis
13	Sum of force terms: pressure, buoyancy, Coriolis and Lorentz
14	Sum of force terms: Lorentz/Coriolis
15	Sum of force terms: Pressure/Lorentz
16	Sum of force terms: Coriolis/Inertia/Archimedean

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

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

8.3.13 dtBrms.TAG

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

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

No. of column	Contents
1	time
2	Changes in magnetic field (poloidal)
3	Changes in magnetic field (toroidal)
4	Poloidal induction term
5	Toroidal induction term
8	Poloidal diffusion term
9	Toroidal diffusion term
10	Omega effect / toroidal induction term
11	Omega effect
12	Production of the dipole field
13	Production of the axisymmetric dipole field

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

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

8.3.14 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	time
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)
```

8.3.15 phase.TAG

This file contains several diagnostic related to phase field whenever this field is used by MagIC. This file is calculated by the subroutine *outPhase*.

No. of column	Contents
1	time
2	Mean spherically-symmetric radius of the solidus
3	Mean temperature of the solidification front
4	Volume of the solid phase
5	Kinetic energy of the solid phase
6	Kinetic energy of the liquid phase
7	Heat flux at the inner core boundary
8	Time variation of of temperature and phase field: $\frac{\partial}{\partial t}(T - St\Phi)$

```
>>> # To stack all the phase.TAG files of the current directory
>>> ts = MagicTs(field='phase', 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
6	Local flow length-scale based on the non-axisymmetric flow components
7	Local flow length-scale based on the peak of the poloidal kinetic energy
8	Standard deviation of magnetic Reynolds number
9	Standard deviation of local Rossby number (mass-weighted)
10	Standard deviation of local Rossby number (RMS velocity)
11	Standard deviation of convective lengthscale
12	Standard deviation of convective lengthscale (non-axi)
13	Standard deviation of convective lengthscale (pol. peak)

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

```
>>> rad = MagicRadial(field='parR')
```

8.4.4 heatR.TAG

Note: This file is **only** written when an equation for the heat transport (temperature or entropy) is solved.

This file contains several time and horizontally averaged thermodynamic properties (temperature, pressure, entropy, etc.) and their variance. This file is calculated by the subroutine *outHeat*.

No. of column	Contents
1	Radial level
2	Entropy (spherically-symetric contribution)
3	Temperature (spherically-symetric contribution)
4	Pressure (spherically-symetric contribution)
5	Density (spherically-symetric contribution)
6	Chemical composition (spherically-symetric contribution)
7	Standard deviation of entropy
8	Standard deviation of temperature
9	Standard deviation of pressure
10	Standard deviation of density
11	Standard deviation of chemical composition

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

```
>>> rad = MagicRadial(field='heatR')
```

8.4.5 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	Chemical power: $Ra_\xi\, g(r)\, \langle u_r \xi' \rangle_s$
4	Viscous dissipation: $\langle (\sigma)^2 \rangle_s$
5	Ohmic dissipation: $\langle (\nabla \times B)^2 \rangle_s$
6	Standard deviation of buoyancy power
7	Standard deviation of chemical power
8	Standard deviation of viscous dissipation
9	Standard deviation of ohmic dissipation

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

```
>>> rad = MagicRadial(field='powerR')
```

8.4.6 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:** $$\mathcal{F}_{poyn} = \frac{Pr\, Di}{Ra\, E\, Pm^2}\langle(\boldsymbol{\nabla}\times\boldsymbol{B})\times\boldsymbol{B}\rangle_s$$
8	Standard deviation of conductive flux
9	Standard deviation of convective flux
10	Standard deviation of kinetic flux
11	Standard deviation of viscous flux
12	Standard deviation of Poynting flux
13	Standard deviation of resistive flux

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

```
>>> rad = MagicRadial(field='fluxesR')
```

8.4.7 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	**horizontal velocity:** $$u_h = \left\langle \sqrt{u_\theta^2 + u_\phi^2} \right\rangle_s$$
4	**radial derivative of the horizontal velocity:** $$\partial u_h / \partial r$$
5	**thermal dissipation rate:** $$\epsilon_T = \langle (\nabla T)^2 \rangle_s$$
6	Standard deviation of entropy
7	Standard deviation of horizontal velocity u_h
8	Standard deviation of the radial derivative of u_h
9	Standard deviation of the thermal dissipation rate

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)
```

8.4.8 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
6	Standard deviation of energy perpendicular to the rotation axis
7	Standard deviation of energy parallel to the rotation axis
8	Standard deviation of axisymmetric energy perpendicular to the rotation axis
9	Standard deviation of axisymmetric energy parallel to the rotation axis

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

```
>>> rad = MagicRadial(field='perpParR')
```

phiR.TAG

This file contains several time-averaged radial profiles related to phase field.

No. of column	Contents
1	radial level
2	Time-averaged spherically-symmetric phase field
3	Standard deviation of spherically-symmetric phase field

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

```
>>> rad = MagicRadial(field='phiR')
```

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 log 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: $\nabla\cdot(K(r)\tilde{T}(r)\nabla s_0)$
9	inverse of the Gruneisen parameter :math`1/Gamma`: $(\partial\ln\tilde{\rho}/\partial\ln\tilde{T})_S$
10	radial derivative of the log of temperature: $\beta = \mathrm{d}\ln\tilde{T}/\mathrm{d}r$

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]_spec_#.TAG and 2D_[kin|mag]_spec_ave.TAG

Note: Those files are **only** written when *l_2D_spectra=.true.*. The time-averaged files also require that *l_spec_avg=.true.*.

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 two kinds of those files that correspond to the aforementioned spectra, namely **2D_kin_spec_#.TAG**, **2D_mag_spec_#.TAG**. In case time-averages are requested, **2D_kin_spec_ave.TAG** and **2D_mag_spec_ave.TAG** will also be stored. 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
```

(continues on next page)

(continued from previous page)

```
!-------------

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 = MagicSpectrum2D(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_spec_avg=.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*.

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	Standard deviation of poloidal kinetic energy versus degree
7	Standard deviation of poloidal kinetic energy versus order
8	Standard deviation of toroidal kinetic energy versus degree
9	Standard deviation of toroidal kinetic energy 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_spec_avg=.true.* and the run is magnetic

This file contains the time-average magnetic energy spectra. This file is written by the subroutine *spectrum*.

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	Standard deviation of the poloidal magnetic energy in the outer core versus degree
9	Standard deviation of the poloidal magnetic energy in the outer core versus order
10	Standard deviation of the toroidal magnetic energy in the outer core versus degree
11	Standard deviation of the toroidal magnetic energy in the outer core versus order
12	Standard deviation of the magnetic energy at the CMB versus degree
13	Standard deviation of the magnetic energy at the CMB 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_spec_avg=.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/order
2	Time-averaged RMS temperature/entropy versus degree
3	Time-averaged RMS temperature/entropy versus order
4	Time-averaged RMS temperature/entropy at the ICB versus degree
5	Time-averaged RMS temperature/entropy at the ICB versus order
6	Time-averaged temperature/entropy gradient at the ICB versus degree
7	Time-averaged temperature/entropy gradient at the ICB versus order
8	Standard deviation of the temperature/entropy versus degree
9	Standard deviation of the temperature/entropy versus order
10	Standard deviation of the temperature/entropy at the ICB versus degree
11	Standard deviation of the temperature/entropy at the ICB versus order
12	Standard deviation of the temperature/entropy gradient at the ICB versus degree
13	Standard deviation of the temperature/entropy gradient at the ICB versus order

8.7.9 dtVrms_spec.TAG

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

This file contains the time-averaged force balance spectra as well as their standard deviation. The calculations are done in the subroutine *dtVrms*.

No. of column	Contents
1	degree + 1
2	Time-averaged Inertia versus degree
3	Time-averaged Coriolis force versus degree
4	Time-averaged Lorentz force versus degree
5	Time-averaged Advection term versus degree
6	Time-averaged Viscous force versus degree
7	Time-averaged thermal Buoyancy versus degree
8	Time-averaged chemical Buoyancy versus degree
9	Time-averaged Pressure gradient versus degree
10 | Time-averaged Pressure/Coriolis balance versus degree	
11	Time-averaged Pressure/Coriolis/Lorentz balance versus degree
12	Time-averaged Pressure/Coriolis/Buoyancy balance versus degree
13	Time-averaged Pressure/Coriolis/Lorentz/Buoyancy balance versus degree
14	Time-averaged Coriolis/Lorentz balance versus degree
15	Time-averaged Pressure/Lorentz balance versus degree
16	Time-averaged Coriolis/Inertia/Buoyancy balance versus degree
17	Standard deviation of Inertia versus degree
18	Standard deviation of Coriolis force versus degree
19	Standard deviation of Lorentz force versus degree
20	Standard deviation of Advection term versus degree
21	Standard deviation of Viscous force versus degree
22	Standard deviation of thermal Buoyancy versus degree
23	Standard deviation of chemical Buoyancy versus degree
24	Standard deviation of Pressure gradient versus degree
25	Standard deviation of Pressure/Coriolis balance versus degree
26	Standard deviation of Pressure/Coriolis/Lorentz balance versus degree
27	Standard deviation of Pressure/Coriolis/Buoyancy balance versus degree
28	Standard deviation of Pressure/Coriolis/Lorentz/Buoyancy balance versus degree
29	Standard deviation of Coriolis/Lorentz balance versus degree
30	Standard deviation of Pressure/Lorentz balance versus degree
31	Standard deviation of Coriolis/Inertia/Buoyancy balance versus degree

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

```
>>> # To read the file ``dtVrms_spec.test``:
>>> sp = MagicSpectrum(field='dtVrms', tag='test')
```

8.7.10 2D force balance spectra `2D_dtVrms_spec.TAG`

Note: Those files are **only** written when *l_RMS=.true.* and *l_2D_RMS=.true.*.

Those files contain 2-D force balance spectra in the (r, ℓ) plane. The calculations are done in the subroutine *dtVrms*. The output file is stored as a Fortran unformatted file.

The structure of the 2D force balance spectra files are as follows:

```
!------------
! Line 1
!------------

version

!-------------
! Line 2
!-------------

n_r_max, l_max ! radial resolution, max(\ell)

!-------------
! Line 3
!-------------

r(1), r(2), r(3), ..., r(n_r_max)                    ! Radius

!-------------
! Line 4
!-------------

Cor_l(l=1,r=1), Cor_l(l=1,r=2), ..., Cor_l(l=1,r=n_r_max),       ! Coriolis␣
↪force
...                                                               ! versus degree
Cor_l(l=l_max,r=1), Cor_l(l=l_max,r=2), ..., Cor_l(l=l_max,r=n_r_max),

!-------------
! Line 5
!-------------

Adv_l ! Advection

!-------------
! Line 6
!-------------

LF_l ! Lorentz force

!-------------
! Line 7
!-------------
```

(continues on next page)

(continued from previous page)

```
Buo_temp_l ! Thermal buoyancy

!------------
! Line 8
!------------

Buo_xi_l ! Chemical buoyancy

!-------------
! Line 9
!-------------

Pre_l ! Pressure

!-------------
! Line 10
!-------------

Dif_l ! Viscosity

!-------------
! Line 11
!-------------

Iner_l ! Inertia

!-------------
! Line 12
!-------------

Geo_l ! Sum of force terms: geostrophic balance

!-------------
! Line 13
!-------------

Mag_l ! Sum of force terms: pressure, Coriolis and Lorentz

!-------------
! Line 14
!-------------

Arc_l ! Sum of force terms: pressure, buoyancy and Coriolis

!-------------
! Line 15
!-------------

ArcMag_l ! Sum of force terms: pressure, buoyancy, Coriolis and Lorentz

!-------------
! Line 16
```

(continues on next page)

(continued from previous page)

```
!-------------

CIA_l ! Sum of force terms Coriolis/Inertia/Archimedean

!-------------
! Line 17
!-------------

CLF_l ! Sum of force terms Coriolis/Lorentz

!-------------
! Line 18
!-------------

PLF_l ! Sum of force terms Pression/Lorentz
```

Those files can be read using the python class *MagicSpectrum2D* with the following options:

```
>>> # Read the file 2D_dtVrms_spec.ext
>>> sp = MagicSpectrum2D(tag='ext', field='dtVrms')
>>> # Print Cor_l
>>> print(sp.Cor_l)
```

8.7.11 2D spectra *am_[kin|mag]_[pol|tor].TAG*

Those files contain the time evolution of the poloidal and toroidal kinetic and magnetic spectra for a given range of spherical harmonic orders m. There are four kinds of those files that correspond to the aforementioned spectra, namely **am_kin_pol.TAG**, **am_kin_tor.TAG**, **am_mag_pol.TAG** and **am_mag_tor.TAG**. The calculations are done in the subroutine *get_amplitude*. The structure of the output files is the same for the four outputs (fortran unformatted files):

```
!-------------
! Line 1
!-------------

time(t=0), e_p_m(m=0,t=0), e_p_m(m=1,t=0), ..., e_p_m(m=m_max_modes,t=0)

...

!-------------
! Line N
!-------------

time(t=N), e_p_m(m=0,t=N), e_p_m(m=1,t=N), ..., e_p_m(m=m_max_modes,t=N)

...
```

Those files can be read using the python class *MagicTs* with the following options:

```
>>> # Read the file am_mag_pol.ext
>>> ts = MagicTs(field='am_mag_pol', tag='ext')
```

(continues on next page)

(continued from previous page)

```
>>> # Print the time
>>> print(ts.time)
>>> # Print the energy content in m=11 for all times
>>> print(ts.coeffs[:, 11])
```

8.8 Graphic files G_#.TAG and G_ave.TAG

These are fortran unformatted files containing 3D data (in the form vector_array(phi, theta, r)) which can be used to visualize the solution. They are written after a fixed number of time steps as specified by the user in the *Output Control namelist* using the parameters listed in the section on *output of graphic files*. In case *l_average* is set to `.true.`, then an average graphic file, named `G_ave.TAG`, containing time averaged values of 3D data, is also written at the end of the simulation.

These files are written in chunks of latitude for one radial level at a time by the subroutine *graphOut* or by *graphOut_mpi* depending on whether `USE_MPI` is set to `Yes` or `No` in the Makefile. The structure of the file looks like below:

```
!-------------
! Line 1
!-------------

version    !Graphout_version_9 (using MPI without comp. without pressure)
           !Graphout_version_10 (using MPI, without comp. with pressure)
           !Graphout_version_11 (using MPI, with comp. without pressure)
           !Graphout_version_12 (using MPI, with comp. with pressure)
           !Graphout_version_5 (without MPI, with pressure and comp.)
           !Graphout_version_6 (without MPI, with comp. without pressure)
           !Graphout_version_7 (without MPI, without comp. without pressure)
           !Graphout_version_8 (without MPI, without comp. with pressure)
!----------
! Line 2
!----------

runid

!----------
! Line 3
!----------

time, n_r_max, n_theta_max, n_phi_tot,          !time = Time of writing
n_r_ic_max-1, minc, nThetasBs,                  !(Simulation time),
ra, ek, pr, prmag,                              !nThetasBs = no. of
radratio, sigma_ratio                           !theta blocks

!----------
! Line 4
!----------

theta(1:n_theta_max)
```

(continues on next page)

(continued from previous page)

```
!------------------------------------------------------------------------

!-----------------------------
!Graphout_version_[9/10/11/12]
!-----------------------------

! These versions are written when the code uses MPI (USE_MPI=yes). Parallel
! chunks of fields are written for different radial levels. Chunks in theta
! are written in parallel using OpenMP

!---------------
! Data
!---------------

!-------------
! Block N
!-------------

!------------
! Line 4 + N
!------------


n_r-1, r(n_r)/r(1), n_theta_start, n_theta_stop   !Radial index, radius in␣
↪terms
                                                  !of r_cmb, start and stop of
                                                  !the theta block

!---------------
! Line 4 + (N+1)
!---------------

sr(1:n_phi_tot, n_theta_start:n_theta_stop, n_r)  !Entropy

!---------------
! Line 4 + (N+2)
!---------------

vr(1:n_phi_tot, n_theta_start:n_theta_stop, n_r)  !Radial velocity

!---------------
! Line 4 + (N+3)
!---------------

vt(1:n_phi_tot, n_theta_start:n_theta_stop, n_r)  !Theta component of velocity

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

vp(1:n_phi_tot, n_theta_start:n_theta_stop, n_r)  !Zonal (phi component) of
                                                  !velocity
```

(continues on next page)

(continued from previous page)

```
if (l_chemical_conv):                                !If composition is stored

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

  xir(1:n_phi_tot, n_theta_start:n_theta_stop, n_r)  !composition



if (l_PressGraph):                                   !If pressure is stored

  !-----------------
  ! Line 4 + (N+6/7)
  !-----------------

  pr(1:n_phi_tot, n_theta_start:n_theta_stop, n_r)  !pressure


if (l_mag):                                           !For a magnetic run

  !-------------------
  ! Line 4 + (N+5/6/7)
  !-------------------

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

  !-------------------
  ! Line 4 + (N+6/7/8)
  !-------------------

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

  !-------------------
  ! Line 4 + (N+7/8/9)
  !-------------------

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


!--------------------------
!Graphout_version_[5/6/7/8]
!--------------------------

!This version is written when the code does not use MPI (USE_MPI=no).
!Chunks in theta are written in parallel with OpenMP.

!---------------
! Data
!---------------
```

(continues on next page)

(continued from previous page)

```
!-------------
! 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
```

(continues on next page)

(continued from previous page)

```
!-----------------------------
! 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_chemical_conv):             !If chemical composition is stored

!-------------
! Composition
!-------------

xi(1,n_theta_start,n_r)           !n_phi = 1, n_theta = n_theta_start, n_r
xi(2,n_theta_start,n_r)           !n_phi = 2, n_theta = n_theta_start, n_r
...
xi(n_phi_tot,n_theta_start,n_r)   !n_phi = n_phi_tot, n_theta = n_theta_start,␣
↪n_r
xi(1,n_theta_start+1,n_r)         !n_phi = 1, n_theta = n_theta_start+1, n_r
```

(continues on next page)

(continued from previous page)

```
...
xi(n_phi_tot,n_theta_start+1,n_r)
...
xi(1,n_theta_stop,n_r)              !n_phi = 1, n_theta = n_theta_stop, n_r
xi(2,n_theta_stop,n_r)              !n_phi = 2, n_theta = n_theta_stop, n_r
...
xi(n_phi_tot,n_theta_stop,n_r)      !n_phi = n_phi_tot, n_theta = n_theta_stop,␣
↪n_r


if (l_PressGraph):                  !If pressure is stored

!-----------
! Pressure
!-----------

pr(1,n_theta_start,n_r)             !n_phi = 1, n_theta = n_theta_start, n_r
pr(2,n_theta_start,n_r)             !n_phi = 2, n_theta = n_theta_start, n_r
...
pr(n_phi_tot,n_theta_start,n_r)     !n_phi = n_phi_tot, n_theta = n_theta_start,␣
↪n_r
pr(1,n_theta_start+1,n_r)           !n_phi = 1, n_theta = n_theta_start+1, n_r
...
pr(n_phi_tot,n_theta_start+1,n_r)
...
pr(1,n_theta_stop,n_r)              !n_phi = 1, n_theta = n_theta_stop, n_r
pr(2,n_theta_stop,n_r)              !n_phi = 2, n_theta = n_theta_stop, n_r
...
pr(n_phi_tot,n_theta_stop,n_r)      !n_phi = n_phi_tot, n_theta = n_theta_stop,␣
↪n_r


if (l_mag):                         !Only if it is a magnetic case

!----------------------
! Radial magnetic field
!----------------------

br(1,n_theta_start,n_r)             !n_phi = 1, n_theta = n_theta_start, n_r
br(2,n_theta_start,n_r)             !n_phi = 2, n_theta = n_theta_start, n_r
...
br(n_phi_tot,n_theta_start,n_r)     !n_phi = n_phi_tot, n_theta = n_theta_start,␣
↪n_r
br(1,n_theta_start+1,n_r)           !n_phi = 1, n_theta = n_theta_start+1, n_r
...
br(n_phi_tot,n_theta_start+1,n_r)
...
br(1,n_theta_stop,n_r)              !n_phi = 1, n_theta = n_theta_stop, n_r
br(2,n_theta_stop,n_r)              !n_phi = 2, n_theta = n_theta_stop, n_r
...
br(n_phi_tot,n_theta_stop,n_r)      !n_phi = n_phi_tot, n_theta = n_theta_stop,␣
↪n_r
```

(continues on next page)

(continued from previous page)

```
!-----------------------------------
! 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*.

```
>>> gr = MagicGraph(ivar = 1, tag='TAG')
>>> # print radial velocity
>>> print(gr.vr)
```

They can be visualized using the *Surf* class:

```
>>> s = Surf(tag='TAG')
>>> # Surface map of radial velocity:
>>> s.surf(field = 'vr', r = 0.5, cm = '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

!-------
```

(continues on next page)

(continued from previous page)

```
! 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
!-----------

!---------------
```

(continues on next page)

(continued from previous page)

```
! 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 checkpoint_*.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
```

(continues on next page)

(continued from previous page)

```
                                                !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
```

The checkpoint files can be read using the python class `MagicCheckpoint`.

```
>>> chk = MagicCheckpoint(filename='checkpoint_end.test')
>>> # print size of poloidal and l_max
>>> print(chk.wpol.shape, chk.l_max)
>>> # convert from cheb to FD using 96 grid points
>>> chk.cheb2fd(96)
>>> write new file
>>> chk.write('checkpoint_fd.test')
```

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 temperature) at specific depths.

In the following, `time(j)` is the time during the j^{th} time step, `time(N)` being the last step. `real` and `imag` denote real and imaginary parts, respectively, of spherical harmonic coefficients. Also, the following notations will be used for the coefficients of potentials (note that scalar fields like temperature do not have a poloidal/toroidal decomposition):

Field	Poloidal	Toroidal
Magnetic	`b`	`aj`
Velocity	`w`	`z`
Temperature	`s`	

First and second derivatives are denoted with a differential notation. e.g: `dw` is the first derivative of `w`, while `ddb` is the second derivative of `b`.

8.11.1 B_coeff_cmb.TAG

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

This file contains time series of spherical harmonic coefficients for the poloidal potential of the magnetic field at the outer boundary (CMB) up to a spherical harmonic degree given by *l_max_cmb*. The detailed calculations are done in the subroutine *write_Bcmb*. The contents of the file look as follows:

- **Header** The file header consists of the information: *l_max_cmb*, *minc* and the number of data points `n_data`.
- **Data** Each chunk of data after the header has the same pattern of `time` followed by a list of real and imaginary values of coefficients.

Thus, on a whole, the structure of the file looks like follows:

```
!------------
! Line 1
!------------

l_max_cmb, minc, n_data

!------------------------------
...

!------------
! Line j + 1
!------------

time(j),
real(b(l=1,m=0)), imag(b(l=1,m=0)),
real(b(l=2,m=0)), imag(b(l=2,m=0)),
...
real(b(l=l_max_cmb,m=l_max_cmb)), imag(b(l=l_max_cmb,m=l_max_cmb)),

...

!-------------
! Line N + 1
!-------------

time(N),
real(b(l=1,m=0)), imag(b(l=1,m=0)),
real(b(l=2,m=0)), imag(b(l=2,m=0)),
...
real(b(l=l_max_cmb,m=l_max_cmb)), imag(b(l=l_max_cmb,m=l_max_cmb))
```

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

```
>>> # To stack the files B_cmb_coeff.testc to B_cmb_coeff.testf
>>> cmb = MagicCoeffCmb(tag='test[c-f]')
>>> # print Gauss coefficient for (\ell=10, m=3)
>>> print(cmb.glm[:, cmb.idx[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(lm=1)), imag(b(lm=1)),
real(b(lm=2)), imag(b(lm=2)),
...
real(b(lm=lm_max)), imag(b(lm=lm_max)),
real(db(lm=1)), imag(db(lm=1)),
real(db(lm=2)), imag(db(lm=2)),
...
real(db(lm=lm_max)), imag(db(lm=lm_max)),
real(aj(lm=1)), imag(aj(lm=1)),
real(aj(lm=2)), imag(aj(lm=2)),
...
real(aj(lm=lm_max)), imag(aj(lm=lm_max)),
real(ddb(lm=1)), imag(ddb(lm=1)),
real(ddb(lm=1)), imag(ddb(lm=1)),
```

(continues on next page)

(continued from previous page)

```
...
real(ddb(lm=lm_max)), imag(ddb(lm=lm_max)),

...

!------------
! Line N + 1
!------------

time(N),
real(b(lm=1)), imag(b(lm=1)),
real(b(lm=2)), imag(b(lm=2)),
...
real(b(lm=lm_max)), imag(b(lm=lm_max)),
real(db(lm=1)), imag(db(lm=1)),
real(db(lm=2)), imag(db(lm=2)),
...
real(db(lm=lm_max)), imag(db(lm=lm_max)),
real(aj(lm=1)), imag(aj(lm=1)),
real(aj(lm=2)), imag(aj(lm=2)),
...
real(aj(lm=lm_max)), imag(aj(lm=lm_max)),
real(ddb(lm=1)), imag(ddb(lm=1)),
real(ddb(lm=1)), imag(ddb(lm=1)),
...
real(ddb(lm=lm_max)), imag(ddb(lm=lm_max)),
```

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[:, cr.idx[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(lm=1)), imag(w(lm=1)),
real(w(lm=2)), imag(w(lm=2)),
...
real(w(lm=lm_max)), imag(w(lm=lm_max)),
real(dw(lm=1)), imag(dw(lm=1)),
real(dw(lm=2)), imag(dw(lm=2)),
...
real(dw(lm=lm_max)), imag(dw(lm=lm_max)),
real(z(lm=1)), imag(z(lm=1)),
real(z(lm=2)), imag(z(lm=2)),
...
real(z(lm=lm_max)), imag(z(lm=lm_max)),

...

!--------------
! Line N + 1
!--------------

time(N),
real(w(lm=1)), imag(w(lm=1)),
real(w(lm=2)), imag(w(lm=2)),
...
real(w(lm=lm_max)), imag(w(lm=lm_max)),
real(dw(lm=1)), imag(dw(lm=1)),
real(dw(lm=2)), imag(dw(lm=2)),
...
real(dw(lm=lm_max)), imag(dw(lm=lm_max)),
real(z(lm=1)), imag(z(lm=1)),
real(z(lm=2)), imag(z(lm=2)),
...
real(z(lm=lm_max)), imag(z(lm=lm_max))
```

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[:, cr.idx[6, 0]], cr.zlm[:, cr.idx[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(lm=0)), imag(s(lm=0)),
real(s(lm=1)), imag(s(lm=1)),
real(s(lm=2)), imag(s(lm=2)),
...
real(s(lm=lm_max)), imag(s(lm=lm_max)),

!------------
! Line N + 1
!------------

time(N),
real(s(lm=0)), imag(s(lm=0)),
real(s(lm=1)), imag(s(lm=1)),
real(s(lm=2)), imag(s(lm=2)),
...
real(s(lm=lm_max)), imag(s(lm=lm_max)),
```

8.12 TO outputs

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

8.12.1 Tay.TAG

This file contains the time series of the Taylorization as well as some measures of the relative geostrophic energy. It is written by the subroutine *outTO*.

No. of column	Contents
1	time
2	Relative fraction of toroidal axisymmetric energy (squared)
3	Relative fraction of geostrophic energy (squared)
4	Taylorization
5	A Taylorization measure based on Reynolds stresses
6	A Taylorization measure based on viscous stresses
7	Total kinetic energy computed on the cylindrical grid (to estimate the accuracy of the method)

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

```
>>> # To load the most recent 'Tay.TAG' file in a directory
>>> ts = MagicTs(tag='Tay')
```

8.12.2 TOnhs.TAG and TOshs.TAG

Those files correspond to the z-averaging of the axisymmetric phi component of the Navier-Stokes equations. It contains the different cylindrical profiles of the forces involved the zonal equation as well as some additional measures of the Taylorization of the solution. shs corresponds to Southern Hemisphere (inside the tangent cylinder), while nhs corresponds to Northern Hemisphere).

Those files can be read using *MagicTOHemi* with the following options:

```
>>> # To load 'TOshs.test' and plot the time-averaged forces:
>>> tos = MagicTOHemi(tag='test', hemi='s', iplot=True)
```

8.12.3 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
!----------
```

(continues on next page)

(continued from previous page)

```
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
!-------------
```

(continues on next page)

(continued from previous page)

```
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|Xi]_lmr_#.TAG`

Those files contain a snapshot of either poloidal/toroidal potentials `V_lmr_#.TAG` and `B_lmr_#.TAG` or a scalar like temperature/entropy or chemical composition (`T_lmr_#.TAG` or `Xi_lmr_#.TAG`) in the radial space for all spherical harmonic degrees and orders. The detailed calculations are done in the subroutine *write_Pot*. The outputs are stored as a fortran unformatted file with a stream access. It has the following structure

```
!-------------
! Header
!-------------
version
time, ra, pr, raxi, sc, prmag, ek, radratio, sigma_ration ! Parameters
n_r_max, n_r_ic_max, l_max, minc, lm_max                  ! Truncation
```

(continues on next page)

(continued from previous page)

```
omega_ic, omega_ma                                          ! Rotation rates
r(1), r(2), ..., r(n_r_max)                                 ! Radius
rho0(1), rho0(2), ..., rho0(n_r_max)                        ! Background density

!-------------
! Poloidal potential or scalar
!-------------
w(lm=1,n_r=1), w(lm=2, n_r=1), ..., w(lm=lm_max, n_r=1),
...
w(lm=1,n_r=n_r_max), ..., w(lm=lm_max,n_r=n_r_max)

!-------------
! If stored: toroidal potential
!-------------
z(lm=1,n_r=1), z(lm=2, n_r=1), ..., z(lm=lm_max, n_r=1),
...
z(lm=1,n_r=n_r_max), ..., z(lm=lm_max,n_r=n_r_max)

!*****************************************************************************!
! This last part is optional and are is written when there is                 !
! an electrically-conducting inner-core                                       !
!*****************************************************************************!
b_ic(lm=1,n_r=1), b_ic(lm=2, n_r=1), ..., b_ic(lm=lm_max, n_r=1),
...
b_ic(lm=1,n_r=n_r_max), ..., b_ic(lm=lm_max,n_r=n_r_max)
aj_ic(lm=1,n_r=1), aj_ic(lm=2, n_r=1), ..., aj_ic(lm=lm_max, n_r=1),
...
aj_ic(lm=1,n_r=n_r_max), ..., aj_ic(lm=lm_max,n_r=n_r_max)
```

The potential files can be read and transformed to the physical space using the python class *MagicPotential*.

```
>>> p = MagicPotential(field='V')
>>> print(p.pol[p.idx[3,2], 32]) # print w(l=3,m=2,n_r=32)
```

Once transformed to the physical space using a Fourier and a Legendre transform, they can be displayed:

```
>>> p.equat(field='vr', cm='jet', levels=50) # Equatorial cut of vr
>>> p.avg(field='vp') # Azimuthal average of vphi
>>> p.surf(field='vt', r=0.8) # Radial cut of vtheta at r=0.8r_o
```

CHAPTER

NINE

DATA VISUALISATION AND POST-PROCESSING

Most of the *output files* written during a run of MagIC can be treated with the python post-processing classes and functions present in the `$MAGIC_HOME/python/magic` directory. These classes depend on several python libraries that can be usually found in most of the Linux distributions.

9.1 Requirements

9.1.1 Hard dependencies

- python 2.7/3.3 or higher.
- matplotlib 1.0 or higher.
- scipy 0.10 or higher.

9.1.2 Optional dependencies

- Although entirely optional, the installation of ipython makes the interactive use of the post-processing python functions much more pleasant. Installing it is therefore recommanded for a smoother interactive usage of the python functions.
- The installation of the basemap toolkit is optional. If installed, additional projections for the *`magic.Surf`* (Aitoff, orthographic, Mollweide, etc.) class will be provided for 2-D surface plotting. Otherwise, the usage of *`magic.Surf`* is limited to the standalone 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`, `Qt5Agg`, `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
```

You can change the default colormap that will be used in the plotting routines.

```
defaultCm = seismic
```

You cant change the default number of contours that will be used in the plotting routines.

```
defaultLevels = 65
```

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 lmrreader_single...
Please wait: building Legendre transforms...
Please wait: building vtklib...
Please wait: building cylavg...
```

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 the **checkpoint_#.TAG** files, see *here*.

7. To read and analyse movie files **_mov.TAG**, see *here*.
8. To read and analyse coeff files **[V|B|T]_coeff.TAG**, see *here*.
9. To read and analyse potential files **[V|B|T]_lmr_#.TAG**, see *here*.
10. To read and analyse radial spectra **B[r|p]Spec.TAG**, see *here*.
11. To read and analyse TO outputs, see *here*.
12. To compare several runs simultaneously, see *here*.
13. To transform the graphic files **G_#.TAG** to a file format readable by paraview, VisIt or mayavi and do some fancy 3-D visualisation, see *here*.
14. For additional diagnostics (boundary layer, heat transport, interpolation on cylindrical grids, etc.), see *here*.
15. 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*
- Geostrophy: *geos.TAG*

- Taylorization measures: *Tay.TAG*
- Heat transfer: *heat.TAG*
- Helicity: *helicity.TAG*
- Velocity square: *u_square.TAG*
- Angular momentum: *AM.TAG*
- Power budget: *power.TAG*
- Earth-likeness of the CMB field: *earth_like.TAG*
- Parallel and perpendicular decomposition: *perpPar.TAG*
- Phase field: *phase.TAG*
- RMS force balance: *dtVrms.TAG*
- RMS induction terms: *dtBrms.TAG*
- Time-evolution of m-spectra: *am_[kin|mag]_[pol|tor].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 ``heat.N0m2z``
>>> ts = MagicTs(field='heat', tag='N0m2z', iplot=False)
```

__init__(*datadir='.'*, *field='e_kin'*, *iplot=True*, *all=False*, *tag=None*)

Parameters

- **datadir** (*str*) – working directory
- **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*, *perpPar=False*, *std=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*, *perpPar=False*, *std=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
- **perpPar** (*bool*) – additional values extracted from *perpPar.TAG* are also computed

__str__()

Formatted output

__weakref__

list of weak references to the object (if defined)

9.3.4 Some resolution/convergence checks

magic.checker.**MagicCheck**(*tstart=None*)

This function is used to compute several sanity checks that can be evaluated if the power.TAG and some spectra have been produced in the current directory. If in addition the tInitAvg file is also there in the directory it averages only from this starting time.

```
>>> MagicCheck(tstart=10.)
```

9.3.5 Support for time-averaged radial profiles

class magic.**MagicRadial**(*datadir='.'*, *field='eKin'*, *iplot=True*, *tag=None*, *tags=None*, *normalize_radius=False*, *quiet=False*)

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*
- Phase field: *phiR.TAG*
- Heat fluxes: *fluxesR.TAG*
- Mean entropy, temperature and pressure: *heatR.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*, *normalize_radius=False*, *quiet=False*)

Parameters

- **datadir** (*str*) – working directory
- **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
- **quiet** (*bool*) – when set to True, makes the output silent (default False)

plot()
Display the result when `iplot=True`

9.3.6 Support for the spectra files `(kin|mag|u2)_spec_#.TAG`

class magic.**MagicSpectrum**(*datadir='.'*, *field='e_kin'*, *iplot=True*, *ispec=None*, *ave=False*, *normalize=False*, *tag=None*, *quiet=False*)
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)
```

__init__(*datadir='.'*, *field='e_kin'*, *iplot=True*, *ispec=None*, *ave=False*, *normalize=False*, *tag=None*, *quiet=False*)

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
- **datadir** (*str*) – current working directory
- **quiet** (*bool*) – when set to True, makes the output silent (default False)

plot()
Plotting function

9.3.7 Support for the 2-D spectra files

class magic.**MagicSpectrum2D**(*datadir='.', field='e_mag', iplot=False, ispec=None, tag=None, cm='jet', levels=33, precision=<class 'numpy.float64'>, ave=False*)
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*
- 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=False, ispec=None, tag=None, cm='jet', levels=33, precision=<class 'numpy.float64'>, ave=False*)

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
- **ave** (*bool*) – plot a time-averaged spectrum when set to True

plot(*levels*, *cm*, *cut=1.0*)
Plotting function

Parameters

- **levels** (*int*) – number of contour levels
- **cm** – name of the colormap
- **cut** (*float*) – adjust the contour maximum to max(abs(data))*cut

9.3.8 Support for G_#.TAG files

class magic.**MagicGraph**(*ivar=None*, *datadir='.'*, *quiet=True*, *ave=False*, *tag=None*, *precision=<class 'numpy.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=np.float64)
```

__init__(*ivar=None*, *datadir='.'*, *quiet=True*, *ave=False*, *tag=None*, *precision=<class 'numpy.float32'>*)

Parameters

- **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 np.float32)

read_record_marker(*filename*, *endian*, *quiet=True*)
This function is used to read a Graphic file that contains record markers.

Parameters

- **filename** (*str*) – name of the graphic file
- **endian** (*str*) – endianness of the file
- **quiet** (*bool*) – when set to True, makes the output silent

read_stream(*filename*, *endian*)
This function is used to read a Graphic file that has no record marker.

Parameters

- **filename** (*str*) – name of the graphic file
- **endian** (*str*) – endianness of the file

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=<class 'numpy.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=np.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=<class 'numpy.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 np.float32 (single)

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

avg(*field='vphi'*, *levels=65*, *cm='seismic'*, *normed=True*, *vmax=None*, *vmin=None*, *cbar=True*, *tit=True*, *pol=False*, *tor=False*, *mer=False*, *merLevels=16*, *polLevels=16*, *ic=False*, *lines=False*)
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 True
- **tor** (*bool*) – diplay the toroidal axisymmetric field contours when set to True
- **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
- **ic** (*bool*) – when set to True, also display the contour levels in the inner core
- **lines** (*bool*) – when set to True, over-plot solid lines to highlight the limits between two adjacent contour levels

equat(*field='vr'*, *levels=65*, *cm='seismic'*, *normed=True*, *vmax=None*, *vmin=None*, *cbar=True*, *tit=True*, *avg=False*, *normRad=False*, *ic=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.
- **ic** (*bool*) – when set to True, also display the contour levels in the inner core

slice(*field='Bphi'*, *lon_0=0.0*, *levels=65*, *cm='seismic'*, *normed=True*, *vmin=None*, *vmax=None*, *cbar=True*, *tit=True*, *grid=False*, *nGridLevs=16*, *normRad=False*, *ic=False*)

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
- **normRad** (*bool*) – when set to True, the contour levels are normalised radius by radius (default is False)
- **ic** (*bool*) – when set to True, also display the contour levels in the inner core

surf(*field='Bphi'*, *proj='hammer'*, *lon_0=0.0*, *r=0.85*, *vmax=None*, *vmin=None*, *lat_0=30.0*, *levels=65*, *cm='seismic'*, *ic=False*, *lon_shift=0*, *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.)
- **lon_shift** (*int*) – translate map in azimuth (in degrees)
- **lon_0** (*float*) – central azimuth (only used with Basemap)
- **lat_0** (*float*) – central latitude (only used with Basemap)
- **tit** (*bool*) – display the title of the figure when set to True
- **tit** – 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.

- **lines** – when set to True, over-plot solid lines to highlight the limits between two adjacent contour levels

9.3.9 Support for checkpoint_#.TAG files

magic.checkpoint.**Graph2Rst**(*gr*, *filename='checkpoint_ave'*)

This function allows to transform an input Graphic file into a checkpoint file format that can be read by MagIC to restart a simulation.

```
>>> # Load a Graphic File
>>> gr = MagicGraph()
>>> # Produce the file checkpoint_ave.from_G
>>> Graph2Rst(gr, filename='checkpoint_ave.from_G')
```

Parameters

- **gr** (magic.MagicGraph) – the input graphic file one wants to convert into a restart file
- **filename** (*str*) – name of the checkpoint file

class magic.checkpoint.**MagicCheckpoint**(*l_read=True*, *filename=None*)

This class allows to manipulate checkpoint files produced by MagIC. It can read it as

```
>>> chk = MagicCheckpoint(filename='checkpoint_end.test')
>>> print(chk.wpol.shape, chk.l_max)
```

This class can also be used to intepolate from FD to Cheb or the opposite >>> chk.cheb2fd(96) >>> chk.write('checkpoint_fd.test')

One can also transform a Graphic file into a checkpoint >>> gr = MagicGraph() >>> chk = MagicCheckpoint(l_read=False) >>> chk.graph2rst(gr)

Finally one can convert checkpoints from XSHELLS >>> chk = MagicCheckpoint(l_read=False) >>> chk.xshells2magic('st0', 161, rscheme='cheb', cond_state='deltaT')

__init__(*l_read=True*, *filename=None*)

Parameters

- **l_read** (*bool*) – a boolean to decide whether one reads a checkpoint or not
- **filename** (*str*) – name of the checkpoint file to be read

__weakref__

list of weak references to the object (if defined)

cheb2fd(*n_r_max*, *fd_stretch=0.3*, *fd_ratio=0.1*)

This routine is used to convert a checkpoint that has a Gauss-Lobatto grid into a finite-difference grid.

Parameters

- **n_r_max** (*int*) – number of radial grid points of the finite difference grid
- **fd_stretch** (*float*) – stretching of the radial grid
- **fd_ratio** (*float*) – ratio of smallest to largest grid spacing

fd2cheb(*n_r_max*)

This routine is used to convert a checkpoint that has finite differences in radius into a Gauss-Lobatto grid.

Parameters **n_r_max** (*int*) – number of radial grid points of the Gauss-Lobatto grid

graph2rst(*gr*, *filename='checkpoint_ave.from_chk'*)

Parameters

- **gr** (magic.MagicGraph) – the input graphic file one wants to convert into a restart file
- **filename** (*str*) – name of the checkpoint file

read(*filename*)
This routine is used to read a checkpoint file.

Parameters **filename** (*str*) – name of the checkpoint file

write(*filename*)
This routine is used to store a checkpoint file. It only stores the state vector not the past quantities required to restart a multistep scheme.

Parameters **filename** (*str*) – name of the checkpoint file

xshells2magic(*xsh_trailing*, *n_r_max*, *rscheme='cheb'*, *cond_state='deltaT'*, *scale_b=1.0*, *filename='checkpoint_end.from_xhells'*)
This routine is used to convert XSHELLS field[U,B,T].xsh_trailing files into a MagIC checkpoint file.

```
>>> chk = MagicCheckPoint()
>>> # Convert field[U,T,B].st1ns_hr2 into a MagIC checkpoint file
>>> chk.xshells2magic('st1ns_hr2', 512, rscheme='fd', cond_state='mixed',
                      scale_b=4.472136e-4)
```

Parameters

- **xsh_trailing** (*str*) – trailing of the field[U,B,T].xsh_trailing files
- **n_r_max** (*int*) – number of radial grid points to be used
- **rscheme** (*str*) – the type of radial scheme ('cheb' or 'fd')
- **cond_state** (*str*) – the type of conducting state: - 'deltaT': fixed temperature contrast - 'mixed': hybrid forcing (STEP1-2 like)
- **scale_b** (*float*) – a rescaling factor for the magnetic field

magic.checkpoint.**get_map**(*lm_max*, *lmax*, *mmax*, *minc*)
This routine determines the look-up tables to convert the indices (l, m) to the single index lm.

Parameters

- **lm_max** (*int*) – total number of lm combinations.
- **lmax** (*int*) – maximum spherical harmonic degree
- **mmax** (*int*) – maximum spherical harmonic order
- **minc** (*int*) – azimuthal symmetry

Returns returns a list of three look-up tables: idx, lm2l, lm2m

Return type list

magic.checkpoint.**get_truncation**(*n_theta_max*, *nalias*, *minc*)
This routine determines l_max, m_max and lm_max from the values of n_theta_max, minc and nalias.

Parameters

- **n_theta_max** (*int*) – number of points along the colatitude
- **nalias** (*int*) – dealiasing paramete (20 is fully dealiased)
- **minc** (*int*) – azimuthal symmetry

Returns returns a list of three integers: l_max, m_max and lm_max

Return type list

magic.checkpoint.**interp_one_field**(*field*, *rold*, *rnew*, *rfac=None*)

This routine interpolates a complex input field from an old radial grid to a new one.

Parameters

- **field** (*numpy.ndarray*) – the field to be interpolated
- **rold** (*numpy.ndarray*) – the old radial grid points
- **rnew** (*numpy.ndarray*) – the new radial grid points
- **rfac** (*numpy.ndarray*) – a rescaling function that depends on the radius

Returns the field interpolated on the new radial grid

Return type numpy.ndarray

9.3.10 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=<class 'numpy.float32'>*, *deminc=True*, *ifield=0*)

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=False)
>>> m2 = Movie(file='AV_mov.testb', iplot=False)
>>> m3 = Movie(file='AV_mov.testc', iplot=False)
>>> # 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=<class 'numpy.float32'>*, *deminc=True*, *ifield=0*)

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, np.float32 for single precision, np.float64 for double precision
- **cut** (*float*) – adjust the contour extrema to max(abs(data))*cut
- **bgcolor** (*str*) – background color of the figure
- **deminc** (*bool*) – a logical to indicate if one wants do get rid of the possible azimuthal symmetry
- **ifield** (*int*) – in case of a multiple-field movie file, you can change the default field displayed using the parameter ifield

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

avgStd(*ifield=0*, *std=False*, *cut=0.5*, *levels=12*, *cmap='RdYlBu_r'*, *ic=False*)
Plot time-average or standard deviation

Parameters

- **ifield** (*int*) – in case of a multiple-field movie file, you can change the default field displayed using the parameter ifield
- **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(*ifield=0*, *cut=0.5*, *levels=12*, *cmap='RdYlBu_r'*, *png=False*, *step=1*, *normed=False*, *dpi=80*, *bgcolor=None*, *deminc=True*, *ic=False*)

Plotting function (it can also write the png files)

Parameters

- **ifield** (*int*) – in case of a multiple-field movie file, you can change the default field displayed using the parameter ifield
- **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
- **deminc** (*bool*) – a logical to indicate if one wants do get rid of the possible azimuthal symmetry

timeLongitude(*ifield=0*, *removeMean=True*, *lat0=0.0*, *levels=12*, *cm='RdYlBu_r'*, *deminc=True*)

Plot the time-longitude diagram (input latitude can be chosen)

Parameters

- **ifield** (*int*) – in case of a multiple-field movie file, you can change the default field displayed using the parameter ifield
- **lat0** (*float*) – value of the latitude
- **levels** (*int*) – number of contour levels
- **cm** (*str*) – name of the colormap
- **deminc** (*bool*) – a logical to indicate if one wants do get rid of the possible azimuthal symmetry
- **removeMean** (*bool*) – remove the time-averaged part when set to True

class magic.**Movie3D**(*file=None*, *step=1*, *lastvar=None*, *nvar='all'*, *nrout=48*, *ratio_out=2.0*, *potExtra=False*, *precision=<class 'numpy.float32'>*)

This class allows to read the 3D movie files *(B|V)_3D_.TAG* and transform them into a series of VTS files `./vtsFiles/B3D_#.TAG` that can be further read using paraview.

```
>>> Movie3D(file='B_3D.TAG')
```

__init__(*file=None*, *step=1*, *lastvar=None*, *nvar='all'*, *nrout=48*, *ratio_out=2.0*, *potExtra=False*, *precision=<class 'numpy.float32'>*)

Parameters

- **file** (*str*) – file name

- **nvar** (*int*) – the number of timesteps of the movie file we want to plot starting from the last line
- **lastvar** (*int*) – the number of the last timestep to be read
- **step** (*int*) – the stepping between two timesteps
- **precision** (*str*) – precision of the input file, np.float32 for single precision, np.float64 for double precision
- **potExtra** (*bool*) – when set to True, potential extrapolation of the magnetic field outside the fluid domain is also computed
- **ratio_out** (*float*) – ratio of desired external radius to the CMB radius. This is is only used when potExtra=True
- **nrout** (*int*) – number of additional radial grid points to compute the potential extrapolation. This is only used when potExtra=True

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

9.3.11 Support for `B_cmb_coeff.TAG` and `(V|B)_coeff_r#.TAG` files

class magic.coeff.**MagicCoeffCmb**(*tag=None*, *datadir='.'*, *ratio_cmb_surface=1*, *scale_b=1*, *iplot=True*, *lCut=None*, *precision=<class 'numpy.float64'>*, *ave=False*, *sv=False*, *quiet=False*)

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[:, cmb.idx[1, 0]]) # time-series of the axisymmetric dipole
>>> plot(cmb.time, cmb.dglmdt[:, cmb.idx[2, 0]]) # Secular variation of the␣
↪quadrupole
>>> # Display the time-evolution of the CMB field
>>> cmb.movieCmb(levels=12, cm='seismic')
>>> # Save the time-evolution of the CMB field
>>> cmb.movieCmb(levels=12, cm='seismic', png=True)
```

__add__(*new*)
Built-in function to sum two cmb files

Note: So far this function only works for two cmb files with the same grid sizes. At some point, we might introduce grid extrapolation to allow any summation/

__init__(*tag=None*, *datadir='.'*, *ratio_cmb_surface=1*, *scale_b=1*, *iplot=True*, *lCut=None*, *precision=<class 'numpy.float64'>*, *ave=False*, *sv=False*, *quiet=False*)
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
- **ave** (*bool*) – load a time-averaged CMB file when set to True
- **sv** (*bool*) – load a dt_b CMB file when set to True
- **quiet** (*bool*) – verbose when toggled to True (default is True)
- **lCut** (*int*) – reduce the spherical harmonic truncation to l <= lCut
- **datadir** (*str*) – working directory

movieCmb(*cut=0.5*, *levels=12*, *cm='RdYlBu_r'*, *png=False*, *step=1*, *normed=False*, *dpi=80*, *bgcolor=None*, *deminc=True*, *removeMean=False*, *precision=<class 'numpy.float64'>*, *contour=False*, *mer=False*)

Plotting function (it can also write the png files)

Parameters

- **levels** (*int*) – number of contour levels
- **cm** (*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
- **deminc** (*bool*) – a logical to indicate if one wants do get rid of the possible azimuthal symmetry
- **precision** (*char*) – single or double precision
- **contour** (*bool*) – also display the solid contour levels when set to True
- **mer** (*bool*) – display meridians and circles when set to True
- **removeMean** (*bool*) – remove the time-averaged part when set to True

plot()

Display some results when iplot is set to True

timeLongitude(*removeMean=True*, *lat0=0.0*, *levels=12*, *cm='RdYlBu_r'*, *deminc=True*, *shtns_lib='shtns'*)

Plot the time-longitude diagram of Br (input latitude can be chosen)

> **Warning:** the python bindings of SHTns are mandatory to use this plotting function!

Parameters

- **lat0** (*float*) – value of the latitude
- **levels** (*int*) – number of contour levels
- **cm** (*str*) – name of the colormap

- **deminc** (*bool*) – a logical to indicate if one wants do get rid of the possible azimuthal symmetry
- **shtns_lib** (*char*) – version of shtns library used: can be either 'shtns' or 'shtns-magic'
- **removeMean** (*bool*) – remove the time-averaged part when set to True

truncate(*lCut*)

Parameters **lCut** (*int*) – truncate to spherical harmonic degree lCut

class magic.coeff.**MagicCoeffR**(*tag*, *ratio_cmb_surface=1*, *scale_b=1*, *iplot=True*, *field='B'*, *r=1*, *precision=<class 'numpy.float64'>*, *lCut=None*, *quiet=False*)

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[:, cr.idx[10, 10]])
```

__init__(*tag*, *ratio_cmb_surface=1*, *scale_b=1*, *iplot=True*, *field='B'*, *r=1*, *precision=<class 'numpy.float64'>*, *lCut=None*, *quiet=False*)

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', 'T' or 'Xi' (magnetic field, velocity field, temperature or composition)
- **r** (*int*) – an integer to characterise which file we want to plot
- **precision** (*str*) – single or double precision
- **lCut** (*int*) – reduce the spherical harmonic truncation to l <= lCut
- **quiet** (*bool*) – verbose when toggled to True (default is True)

fft()

Fourier transform of the poloidal energy

movieRad(*cut=0.5*, *levels=12*, *cm='RdYlBu_r'*, *png=False*, *step=1*, *normed=False*, *dpi=80*, *bgcolor=None*, *deminc=True*, *removeMean=False*, *precision=<class 'numpy.float64'>*, *contour=False*, *mer=False*)

Plotting function (it can also write the png files)

Parameters

- **levels** (*int*) – number of contour levels
- **cm** (*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
- **deminc** (*bool*) – a logical to indicate if one wants do get rid of the possible azimuthal symmetry
- **precision** (*char*) – single or double precision
- **contour** (*bool*) – also display the solid contour levels when set to True
- **mer** (*bool*) – display meridians and circles when set to True
- **removeMean** (*bool*) – remove the time-averaged part when set to True

truncate(*lCut*, *field='B'*)

Parameters

- **lCut** (*int*) – truncate to spherical harmonic degree lCut
- **field** (*char*) – name of the field ('V', 'B' or 'T')

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

magic.coeff.**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

9.3.12 Support for B[rp]Spec.TAG

class magic.**MagicRSpec**(*tag*, *field='Br'*, *precision=<class 'numpy.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=<class 'numpy.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. np.float32)
- **avg** (*bool*) – when set to True, display time averaged quantities

plotAvg()

Plotting function for time-averaged profiles

9.3.13 Support for [V|B|T]_lmr_#.TAG

class magic.**MagicPotential**(*field='V'*, *datadir='.'*, *tag=None*, *ave=False*, *ipot=None*, *precision=<class 'numpy.float32'>*, *verbose=True*, *ic=False*)

This class allows to load and display the content of the potential files: *V_lmr.TAG*, *B_lmr.TAG* and *T_lmr.TAG*. This class allows to transform the poloidal/toroidal potential in spectral space to the physical quantities in the physical space. It allows to plot radial and equatorial cuts as well as phi-averages.

```
>>> # To read T_lmr.test
>>> p = MagicPotential(field='T', ipot=1, tag='test')
>>> # To read the latest V_lmr file in the working directory
>>> p = MagicPotential(field='V')
>>> # Get the poloidal potential (lm, nR)
>>> wlm = p.pol
>>> # Obtain the value of w(l=12, m=12, nR=33)
>>> print( p.pol[p.idx[12,12], 32] )
```

```
>>> # Possible plots
>>> p.equat(field='vr')
>>> p.avg(field='vp')
>>> p.surf(field='vt', r=0.8)
```

__init__(*field='V'*, *datadir='.'*, *tag=None*, *ave=False*, *ipot=None*, *precision=<class 'numpy.float32'>*, *verbose=True*, *ic=False*)

Parameters

- **field** (*str*) – 'B', 'V', 'T' or 'Xi' (magnetic field, velocity field, temperature or chemical composition)
- **datadir** (*str*) – the working directory

- **tag** (*str*) – if you specify a pattern, it tries to read the corresponding files
- **ave** (*bool*) – plot a time-averaged spectrum when set to True
- **ipot** (*int*) – the number of the lmr file you want to plot
- **precision** (*str*) – single or double precision
- **verbose** (*bool*) – some info about the SHT layout
- **ic** (*bool*) – read or don't read the inner core

avg(*field='vphi'*, *levels=65*, *cm='seismic'*, *normed=True*, *vmax=None*, *vmin=None*, *cbar=True*, *tit=True*)
Plot the azimutal average of a given field.

```
>>> p = MagicPotential(field='V')
>>> # Axisymmetric zonal flows, 65 contour levels
>>> p.avg(field='vp', levels=65, cm='seismic')
```

```
>>> # Minimal plot (no cbar, not title)
>>> p.avg(field='vr', tit=False, cbar=False)
```

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.

equat(*field='vr'*, *levels=65*, *cm='seismic'*, *normed=True*, *vmax=None*, *vmin=None*, *cbar=True*, *tit=True*, *normRad=False*)
Plot the equatorial cut of a given field

```
>>> p = MagicPotential(field='B')
>>> # Equatorial cut of the Br
>>> p.equat(field='Br')
```

```
>>> # Normalise the contour levels radius by radius
>>> p.equat(field='Bphi', normRad=True)
```

Parameters

- **field** (*str*) – the name of the input physical quantity you want to display
- **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.

read(*filename*, *field*, *endian*, *record_marker*, *ic=False*, *precision=<class 'numpy.float32'>*)
This routine defines a reader for the various versions of the lmr files.

Parameters

- **filename** (*str*) – name of the input lmr file
- **field** (*str*) – 'B', 'V', 'T' or 'Xi' (magnetic field, velocity field, temperature or chemical composition)
- **endian** (*str*) – a character string that specifies the endianness of the input file ('B' for big endian or 'l' for little endian)
- **record_marker** (*bool*) – a boolean to specify whether the file contains record marker
- **ic** (*bool*) – read or don't read the inner core
- **precision** (*str*) – single or double precision

surf(*field='vr'*, *proj='hammer'*, *lon_0=0.0*, *r=0.85*, *vmax=None*, *vmin=None*, *lat_0=30.0*, *levels=65*, *cm='seismic'*, *lon_shift=0*, *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).

```
>>> p = MagicPotential(field='V')
>>> # Radial flow component at ``r=0.95 r_o``, 65 contour levels
>>> p.surf(field='vr', r=0.95, levels=65, cm='seismic')
```

```
>>> # Control the limit of the colormap from -1e3 to 1e3
>>> p.surf(field='vp', r=1., vmin=-1e3, vmax=1e3, levels=33)
```

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.
- **lon_0** (*float*) – central azimuth (only used with Basemap)
- **lat_0** (*float*) – central latitude (only used with Basemap)
- **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.

9.3.14 Support for TO outputs

class magic.**TOMovie**(*file=None*, *iplot=True*, *cm='seismic'*, *cut=0.5*, *levels=33*, *avg=True*, *precision=<class 'numpy.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')
```

__add__(*new*)

Built-in function to sum two TO movies

Note: So far this function only works for two TO movies with the same grid sizes. At some point, we might introduce grid extrapolation to allow any summation.

__init__(*file=None*, *iplot=True*, *cm='seismic'*, *cut=0.5*, *levels=33*, *avg=True*, *precision=<class 'numpy.float32'>*)

Parameters

- **file** (*str*) – the filename of the TO_mov file
- **cm** (*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, np.float32 for single precision, np.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

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

class magic.**MagicTOHemi**(*datadir='.'*, *hemi='n'*, *tag=None*, *precision=<class 'numpy.float32'>*, *iplot=False*)

This class can be used to read and display z-integrated quantities produced by the TO outputs. Those are basically the TO[s|n]hn.TAG files

```
>>> to = MagicTOHemi(hemi='n', iplot=True) # For the Northern hemisphere
```

__init__(*datadir='.'*, *hemi='n'*, *tag=None*, *precision=<class 'numpy.float32'>*, *iplot=False*)

Parameters

- **datadir** (*str*) – current working directory
- **hemi** (*str*) – Northern or Southern hemisphere ('n' or 's')
- **tag** (*str*) – file suffix (tag), if not specified the most recent one in the current directory is chosen
- **precision** (*str*) – single or double precision
- **iplot** (*bool*) – display the output plot when set to True (default is True)

plot()

Plotting function

9.3.15 Run comparison

class magic.**CompSims**(*file='liste'*, *field='ts'*, *ncol=4*, *cm='RdYlBu_r'*, *dpi=96*, *normed=True*, *levels=16*, *type=None*, *fullPath=False*, *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*, *fullPath=False*, *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.
- **fullPath** (*bool*) – set to True if the full path is specified in the input file
- **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.

9.3.16 Conversion of G_#.TAG files to vts/vti files

class magic.graph2vtk.**Graph2Vtk**(*gr*, *filename='out'*, *scals=['vr', 'emag', 'tfluct']*, *vecs=['u', 'B']*, *potExtra=False*, *ratio_out=2*, *nrout=32*, *deminc=True*, *outType='vts'*, *nFiles=1*, *nx=96*, *ny=96*, *nz=96*, *labFrame=False*)

This class allows to transform an input graphic file to a file format readable by paraview/visit or mayavi. It also allows to compute a possible potential extrapolation of the field lines in an arbitrary outer spherical shell domain

```
>>> # Load a graphic file
>>> gr = MagicGraph(ivar=1)
>>> # store myOut.vts
>>> Graph2Vtk(gr, 'myOut', outType='vts')
>>> # store u' and B for the vector fields and vortz and T for the scalars
>>> Graph2Vtk(gr, scals=['temp', 'vortz'], vecs=['ufluct', 'B'])
>>> # store only T'
>>> Graph2Vtk(gr, scals=['tempfluct'], vecs=[])
>>> # store only B with its potential extrapolation up to 3*r_cmb
>>> Graph2Vtk(gr, scals=[], vecs=['B'], potExtra=True, ratio_out=3)
>>> # Extrapolate on a cartesian grid of size 128^3
>>> Graph2Vtk(gr, outType='vti', nx=128, ny=128, nz=128)
```

__init__(*gr*, *filename='out'*, *scals=['vr', 'emag', 'tfluct']*, *vecs=['u', 'B']*, *potExtra=False*, *ratio_out=2*, *nrout=32*, *deminc=True*, *outType='vts'*, *nFiles=1*, *nx=96*, *ny=96*, *nz=96*, *labFrame=False*)

Parameters

- **filename** (*str*) – the file name of the output (without extension)
- **gr** (magic.MagicGraph) – the input graphic file one wants to transform to vts/vti
- **scals** (*list(str)*) – a list that contains the possible input scalars: 'entropy', 'vr', 'vp', 'tfluct', 'vortz', 'vortzfluct', 'ekin', 'emag', 'vortr', 'colat'
- **vecs** (*list(str)*) – a list that contains the possible input vectors: 'u', 'b', 'ufluct', 'bfluct'
- **potExtra** (*bool*) – when set to True, calculates the potential extrapolation of the magnetic field up to ratio_out*r_cmb
- **ratio_out** (*float*) – in case of potential extrapolation, this is the ratio of the external outer radius to r_cmb (rout/rcmb)
- **nrout** (*integer*) – in case of potential extrapolation, this input allows to specify thenumber of radial grid points in the outer spherical envelope
- **deminc** (*bool*) – a logical to indicate if one wants do get rid of the possible azimuthal symmetry
- **outType** (*str*) – nature of the VTK file produced. This can be either 'vts' for the spherical grid or 'vti' for an extrapolation on a cartesian grid
- **nFiles** (*int*) – number of output chunks in case of parallel vts file format (pvts)
- **nx** (*int*) – number of grid points in the x direction
- **ny** (*int*) – number of grid points in the x direction
- **nz** (*int*) – number of grid points in the x direction
- **labFrame** (*bool*) – when set to True, transform the velocity to the lab frame

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

writeVTI(*filename*, *nx=96*, *ny=96*, *nz=96*)
In this case, the output is extrapolated on a cartesian grid and then written in a vti file.

Parameters

- **filename** (*str*) – the file name of the output (without extension)
- **nx** (*int*) – number of grid points in the x direction
- **ny** (*int*) – number of grid points in the x direction
- **nz** (*int*) – number of grid points in the x direction

writeVTS(*filename*, *nFiles*)
This function stores the output on a structured-grid vts file.

Parameters

- **filename** (*str*) – the file name of the output (without extension)
- **nFiles** (*int*) – number of outpute files (in case of pvts)

magic.graph2vtk.**sph2cart_scal**(*scals*, *radius*, *nx=96*, *ny=96*, *nz=96*, *minc=1*)
This function interpolates a series of scalar fields from the spherical coordinates to the cartesian coordinates.

Parameters

- **scals** (*numpy.ndarray[nscals,nphi,ntheta,nr]*) – an array that contains the different scalar quantities
- **radius** (*numpy.ndarray*) – the input radius
- **nx** (*int*) – number of grid points in the x direction
- **ny** (*int*) – number of grid points in the x direction
- **nz** (*int*) – number of grid points in the x direction
- **minc** (*int*) – azimuthal symmetry

Returns a tuple that contains the scalars, the max of the grid and the grid spacing

Return type (numpy.ndarray[nscals,nz,ny,nx],float,float)

magic.graph2vtk.**sph2cart_vec**(*vecr*, *vect*, *vecp*, *radius*, *nx=96*, *ny=96*, *nz=96*, *minc=1*)
This function interpolates a series of vector fields from the spherical coordinates to the cartesian coordinates.

Parameters

- **vecr** (*numpy.ndarray[nvecs,nphi,ntheta,nr]*) – the radial components of the different vector fields
- **vect** (*numpy.ndarray[nvecs,nphi,ntheta,nr]*) – the latitudinal components of the different vector fields
- **vecp** (*numpy.ndarray[nvecs,nphi,ntheta,nr]*) – the azimuthal components of the different vector fields
- **radius** (*numpy.ndarray*) – the input radius
- **nx** (*int*) – number of grid points in the x direction
- **ny** (*int*) – number of grid points in the x direction
- **nz** (*int*) – number of grid points in the x direction

- **minc** (*int*) – azimuthal symmetry

Returns a tuple that contains the three vectors components

Return type (numpy.ndarray[nvecs,nz,ny,nx],...)

9.3.17 Potential extrapolation

class magic.potExtra.**ExtraPot**(*rcmb*, *brcmb*, *minc*, *ratio_out=2.0*, *nrout=32*, *cutCMB=False*, *deminc=True*)

This class is used to compute the potential field extrapolation of the magnetic field in an arbitrary outer spherical shell domain. It takes as an input the magnetic field at the CMB.

__init__(*rcmb*, *brcmb*, *minc*, *ratio_out=2.0*, *nrout=32*, *cutCMB=False*, *deminc=True*)

Parameters

- **bcmb** (*numpy.ndarary*) – the surface radial field, array of dimension [np, nt]
- **rcmb** (*float*) – the value of the radius at the surface
- **minc** (*int*) – azimuthal symmetry
- **ratio_out** (*float*) – the ratio of the outer sphere radius to the surface radius
- **nrout** (*int*) – the number of radial point (linearly spaced) of the extrapolated field in the outer spherical domain
- **cutCMB** (*bool*) – a logical if one wants to remove the first grid point (useful if one then wants to merge the graphic file with the extrapolation)
- **deminc** (*bool*) – a logical to indicate if one wants do get rid of the possible azimuthal symmetry

__weakref__

list of weak references to the object (if defined)

avg(*field='br'*, *levels=12*, *cm='RdYlBu_r'*, *normed=True*, *vmax=None*, *vmin=None*)

A small routine to plot the azimuthal averages of the extrapolated fields.

Parameters

- **field** (*str*) – the quantity you want to plot: 'br' or 'bp'
- **levels** (*int*) – the number of contour levels
- **cm** (*str*) – the name of the colormap
- **vmax** (*float*) – maximum value of the contour levels
- **vmin** (*float*) – minimum value of the contour levels
- **normed** (*bool*) – when set to True, the colormap is centered around zero. Default is True, except for entropy/temperature plots.

9.3.18 Additional possible analyses

class magic.bLayers.**BLayers**(*iplot=False*, *quiet=False*)

This class allows to determine the viscous and thermal boundary layers using several classical methods (slope method, peak values, dissipation rates, etc.). It uses the following files:

- Kinetic energy: *eKinR.TAG*
- Power budget: *powerR.TAG*
- Radial profiles used for boundary layers: *bLayersR.TAG*

This function can thus **only** be used when both *powerR.TAG* and *bLayersR.TAG* exist in the working directory.

> **Warning:** This function works well as long as rigid boundaries and fixed temperature boundary conditions are employed. Other combination of boundary conditions (fixed fluxes and/or stress-free) might give wrong results, since boundary layers become awkward to define in that case.

Since this function is supposed to use time-averaged quantities, the usual procedure is first to define the initial averaging time using *AvgField*: (this needs to be done only once)

```
>>> a = AvgField(tstart=2.58)
```

Once the `tInitAvg` file exists, the boundary layer calculation can be done:

```
>>> bl = BLayers(iplot=True)
>>> # print the formatted output
>>> print(bl)
```

__init__(*iplot=False*, *quiet=False*)

Parameters

- **iplot** (*bool*) – display the result when set to True (default False)
- **quiet** (*bool*) – less verbose when set to True (default is False)

__str__()

Formatted output

plot()

Plotting function

magic.bLayers.**getAccuratePeaks**(*rad*, *uh*, *uhTop*, *uhBot*, *ri*, *ro*)

This functions performs a spline extrapolation around the maxima of the input array uh to define a more accurate location of the boundary layer.

Parameters

- **rad** (*numpy.ndarray*) – radius
- **uh** (*numpy.ndarray*) – the horizontal velocity profile
- **uhTop** (*float*) – first peak value of uh close to the outer boundary
- **uhBot** (*float*) – first peak value of uh close to the inner boundary
- **ri** (*float*) – the inner core radius
- **ro** (*float*) – the outer core radius

Returns four floats: thickness of the bottom boundary layer, thickness of the top boundary layer, extrapolated value of uh at the bottom boundary layer, extrapolated value of uh at the top boundary layer

Return type list

magic.bLayers.**getMaxima**(*field*)

This function determines the local maxima of the input array field

Parameters **field** (*numpy.ndarray*) – the input array

Returns a list containing the indices of the local maxima

Return type list

magic.bLayers.**integBotTop**(*rad*, *field*, *ri*, *ro*, *lambdai*, *lambdao*, *normed=False*)

This function evaluates the radial integral of the input array field in the bottom and top boundary layers separately.

Parameters

- **rad** (*numpy.ndarray*) – radius
- **field** (*numpy.ndarray*) – the input radial profile
- **ri** (*float*) – the inner core radius
- **ro** (*float*) – the outer core radius
- **lambdai** (*float*) – thickness of the inner boundary layer
- **lambdao** (*float*) – thickness of the outer boundary layer
- **normed** (*bool*) – when set to True, the outputs are normalised by the volumes of the boundary layers. In that case, the outputs are volume-averaged quantities.

Returns two floats that contains the bottom and top boundary layers integrations (integBot, integTop)

Return type list

magic.bLayers.**integBulkBc**(*rad*, *field*, *ri*, *ro*, *lambdai*, *lambdao*, *normed=False*)

This function evaluates the radial integral of the input array field in the boundary layer and in the bulk separately.

Parameters

- **rad** (*numpy.ndarray*) – radius
- **field** (*numpy.ndarray*) – the input radial profile
- **ri** (*float*) – the inner core radius
- **ro** (*float*) – the outer core radius
- **lambdai** (*float*) – thickness of the inner boundary layer
- **lambdao** (*float*) – thickness of the outer boundary layer
- **normed** (*bool*) – when set to True, the outputs are normalised by the volumes of the boundary layers and the fluid bulk, respectively. In that case, the outputs are volume-averaged quantities.

Returns two floats that contains the boundary layer and the bulk integrations (integBc, integBulk)

Return type list

class magic.**ThetaHeat**(*iplot=False*, *angle=10*, *pickleName='thHeat.pickle'*)

This class allows to conduct some analysis of the latitudinal variation of the heat transfer. It relies on the movie files *ATmov.TAG* and *AHF_mov.TAG*. As it's a bit time-consuming, the calculations are stored in a python.pickle file to quicken future usage of the data.

This function can **only** be used when *bLayersR.TAG* exist in the working directory.

Since this function is supposed to use time-averaged quantities, the usual procedure is first to define the initial averaging time using *AvgField*: (this needs to be done only once)

```
>>> a = AvgField(tstart=2.58)
```

Once the tInitAvg file exists, the latitudinal heat transfer analysis can be done using:

```
>>> # For chunk-averages over 10^\degree in the polar and equatorial regions.
>>> th = ThetaHeat(angle=10)
>>> # Formatted output
>>> print(th)
```

__init__(*iplot=False*, *angle=10*, *pickleName='thHeat.pickle'*)

Parameters

- **iplot** (*bool*) – a boolean to toggle the plots on/off
- **angle** (*float*) – the integration angle in degrees

PickleName calculations a

__str__()
Formatted outputs

```
>>> th = ThetaHeat()
>>> print(th)
```

plot()
Plotting function

class magic.cyl.**Cyl**(*ivar=1*, *datadir='.'*, *ns=None*)
This class allows to extrapolate a given *graphic file* on a cylindrical grid. Once done, the extrapolated file is stored in a python.pickle file. It is then possible to display 2-D cuts of the extrapolated arrays (radial cuts, phi-averages, equatorial cuts, z-averages and phi-slices)

Warning: This process is actually **very demanding** and it might take a lot of time to extrapolate the G_#.TAG file. Be careful when choosing the input value of ns!

```
>>> # Extrapolate the G file to the cylindrical grid (ns=128, nz=2*ns)
>>> c = Cyl(ivar=1, ns=128)
>>> # Radial cut of v_r
>>> c.surf(field='vr', r=0.8)
>>> # Vertical average of B_\phi
>>> c.avgz(field='Bphi', cm='seismic', levels=33)
>>> # Azimuthal average of v_\phi
>>> c.avg(field='Bphi')
>>> # Equatorial cut of of v_theta
>>> c.equat(field='vtheta')
```

__init__(*ivar=1*, *datadir='.'*, *ns=None*)

Parameters

- **ivar** (*int*) – the number of the Graphic file
- **datadir** (*str*) – working directory
- **ns** (*int*) – number of grid points in the radial direction

avg(*field='Bphi'*, *levels=16*, *cm='RdYlBu_r'*, *normed=True*, *vmax=None*, *vmin=None*)
Plot the azimutal average of a given field.

```
>>> c = Cyl(ns=65)
>>> # Azimuthal average of B_r
>>> c.avg(field='Br', cm='seismic', levels=33)
```

Parameters

- **field** (*str*) – name of the input field
- **levels** (*int*) – number of contour levels
- **cm** (*str*) – name of the color map
- **normed** (*bool*) – when set to True, the contours are normalised fro -max(field), max(field)
- **vmin** (*float*) – truncate the contour levels to values > vmin
- **vmax** (*float*) – truncate the contour levels to values < vmax

avgz(*field='vs'*, *levels=16*, *cm='RdYlBu_r'*, *normed=True*, *vmin=None*, *vmax=None*, *avg=False*)
Plot the vertical average of a given field.

```
>>> c = Cyl(ns=65)
>>> # Vertical average of v_s
>>> c.avg(field='vs', cm='seismic', levels=33)
```

Parameters

- **field** (*str*) – name of the input field
- **levels** (*int*) – number of contour levels
- **cm** (*str*) – name of the color map
- **normed** (*bool*) – when set to True, the contours are normalised fro -max(field), max(field)
- **vmin** (*float*) – truncate the contour levels to values > vmin
- **vmax** (*float*) – truncate the contour levels to values < vmax
- **avg** (*bool*) – when set to True, an additional figure with the phi-average profile is also displayed

equat(*field='vs'*, *levels=16*, *cm='RdYlBu_r'*, *normed=True*, *vmax=None*, *vmin=None*)
Plot an input field in the equatorial plane.

```
>>> c = Cyl(ns=65)
>>> # Equatorial cut of v_\phi
>>> c.equat(field='vphi', cm='seismic', levels=33)
```

Parameters

- **field** (*str*) – name of the input field

- **levels** (*int*) – number of contour levels
- **cm** (*str*) – name of the color map
- **normed** (*bool*) – when set to True, the contours are normalised fro -max(field), max(field)
- **vmin** (*float*) – truncate the contour levels to values > vmin
- **vmax** (*float*) – truncate the contour levels to values < vmax

slice(*field='Bphi'*, *lon_0=0.0*, *levels=16*, *cm='RdYlBu_r'*, *normed=True*)
Plot an azimuthal slice of a given field.

```
>>> c = Cyl(ns=65)
>>> # Slices of v_r at 30 and 60 degrees
>>> c.slice(field='vr', lon_0=[30, 60])
```

Parameters

- **field** (*str*) – name of the input field
- **lon_0** (*float or list*) – the longitude of the slice in degrees, or a list of longitudes
- **levels** (*int*) – number of contour levels
- **cm** (*str*) – name of the color map
- **normed** (*bool*) – when set to True, the contours are normalised fro -max(field), max(field)

surf(*field='Bphi'*, *r=0.85*, *vmin=None*, *vmax=None*, *levels=16*, *cm='RdYlBu_r'*, *normed=True*, *figsize=None*)
Plot the surface distribution of an input field at a given input radius (normalised by the outer boundary radius).

```
>>> c = Cyl(ns=65)
>>> # Surface plot of B_\phi from -10 to 10
>>> c.surf(field='Bphi', r=0.6, vmin=-10, vmax=10, levels=65)
```

Parameters

- **field** (*str*) – name of the input field
- **r** (*float*) – radial level (normalised to the outer boundary radius)
- **levels** (*int*) – number of contour levels
- **cm** (*str*) – name of the color map
- **normed** (*bool*) – when set to True, the contours are normalised fro -max(field), max(field)
- **vmin** (*float*) – truncate the contour levels to values > vmin
- **vmax** (*float*) – truncate the contour levels to values < vmax

magic.cyl.**sph2cyl**(*g*, *ns=None*, *nz=None*)
This function interpolates the three flow (or magnetic field) component of a *G_#.TAG* file on a cylindrical grid of size (ns, nz).

Warning: This might be really slow!

Parameters

- **g** (*magic.MagicGraph*) – input graphic output file
- **ns** (*int*) – number of grid points in the radial direction
- **nz** (*int*) – number of grid points in the vertical direction

Returns a python tuple of five numpy.ndarray (S,Z,vs,vp_cyl,vz). S[nz,ns] is a meshgrid that contains the radial coordinate. Z[nz,ns] is a meshgrid that contains the vertical coordinate. vs[nz,ns] is the radial component of the velocity (or magnetic field), vp_cyl[nz,ns] the azimuthal component and vz[nz,ns] the vertical component.

Return type tuple

magic.cyl.**sph2cyl_plane**(*data*, *rad*, *ns*, *nz*)

This function extrapolates a phi-slice of a spherical shell on a cylindrical grid

```
>>> # Read G_1.test
>>> gr = MagicGraph(ivar=1, tag='test')
>>> # phi-average v_\phi and s
>>> vpm = gr.vphi.mean(axis=0)
>>> sm = gr.entropy.mean(axis=0)
>>> # Interpolate on a cylindrical grid
>>> Z, S, 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=<class 'numpy.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=<class 'numpy.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 one wants 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, np.float32 for single precision, np.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

9.3.19 Spectral tansforms

class magic.spectralTransforms.**SpectralTransforms**(*l_max=32*, *minc=1*, *lm_max=561*, *n_theta_max=64*, *verbose=True*)

This python class is used to compute Legendre and Fourier transforms from spectral to physical space. It works in two steps: one first needs to initialize the transform

```
>>> sh = SpectralTransforms( l_max=256, lm_max=33153, n_theta_max=384)
>>> print(Tlm[:, 10].shape) # lm_max (Temperature at ir=10)
>>> T = sh.spec_spat(Tlm) # T[n_phi_max, n_theta_max]
```

spat_spec(**args*)

This subroutine computes a transfrom from spatial representation (n_phi,n_theta) to spectral representation (lm_max). It returns one complex 1-D array (dimension(n_phi_max))

```
>>> gr = MagicGraph()
>>> sh = SpectralTransforms(gr.l_max, gr.minc, gr.lm_max, gr.n_theta_max)
>>> vr = gr.vr[:,:,30] # Radius ir=30
>>> vrlm = sh.spat_spec(vr) # vrlm is a complex array (lm_max)
>>> # Caculation of the poloidal potential from vr:
>>> wlm = np.zeros_like(vrlm)
>>> wlm[1:] = vrlm[1:]/(sh.ell[1:]*(sh.ell[1:]+1))*gr.radius[30]**2
```

(continues on next page)

(continued from previous page)

```
>>> # Spheroidal/Toroidal transform
>>> vtlm, vplm = spec_spat(gr.vtheta, gr.vphi)
```

Parameters **input** (*numpy.ndarray*) – input array in the physical space (n_phi,n_theta)

Returns output array in the spectral space (lm_max)

Return type numpy.ndarray

spec_spat(**args*, ***kwargs*)

This subroutine computes a transfrom from spectral to spatial for all latitudes. It returns either one or two 2-D arrays (dimension(n_phi_max,n_theta_max)) depending if only the poloidal or both the poloidal and the toroidal potentials are given as input quantities.

```
>>> print(wlmr.shape) # lm_max
>>> vr = spec_spat_equat(wlmr)
>>> print(vr.shape) # n_phi, n_theta
>>> vt, vp = spec_spat_equat(dwdrlmr, zlmr)
```

spec_spat_dphi(*polo*)

This routine computes the phi-derivative and the transform from spectral to spatrial spaces. It returns a 2-D array of dimension (n_phi,n_theta)

```
>>> p = MagicPotential('V')
>>> vrlm = p.pol*p.ell*(p.ell+1)/p.radius[ir]**2/p.rho0[ir] # vr at r=ir
>>> dvrdp = p.sh.spec_spat_dphi(vrlm) # phi-derivative of vr
```

Parameters **polo** (*numpy.ndarray*) – the input array(lm_max) in spectral space

Returns the phi derivative in the physical space (n_phi, n_theta)

Return type numpy.ndarray

spec_spat_dtheta(*polo*, *l_axi=False*)

This routine computes the theta-derivative and the transform from spectral to spatrial spaces. It returns a 2-D array of dimension (n_phi,n_theta)

```
>>> p = MagicPotential('V')
>>> vrlm = p.pol*p.ell*(p.ell+1)/p.radius[ir]**2/p.rho0[ir] # vr at r=ir
>>> dvrdt = p.sh.spec_spat_dtheta(vrlm) # theta-derivative of vr
```

Parameters

- **polo** (*numpy.ndarray*) – the input array(lm_max) in spectral space
- **l_axi** (*bool*) – switch to True, if only the axisymmetric field is needed

Returns the theta derivative in the physical space (n_phi, n_theta)

Return type numpy.ndarray

spec_spat_equat(**args*)

This subroutine computes a transfrom from spectral to spatial at the equator. It returns either one or two 1-D arrays (dimension(n_phi_max)) depending if only the poloidal or both the poloidal and the toroidal potentials are given as input quantities.

```
>>> print(wlmr.shape) # lm_max
>>> vr = spec_spat_equat(wlmr)
>>> print(vr.shape) # n_phi
>>> vt, vp = spec_spat_equat(dwdrlmr, zlmr)
```

9.3.20 Plotting functions

magic.plotlib.**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.plotlib.**equatContour**(*data*, *radius*, *minc=1*, *label=None*, *levels=65*, *cm='seismic'*, *normed=True*, *vmax=None*, *vmin=None*, *cbar=True*, *tit=True*, *normRad=False*, *deminc=True*, *bounds=True*)

Plot the equatorial cut of a given field

Parameters

- **data** (*numpy.ndarray*) – the input data (an array of size (nphi,nr))
- **radius** (*numpy.ndarray*) – the input radius
- **minc** (*int*) – azimuthal symmetry
- **label** (*str*) – the name of the input physical quantity you want to display
- **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.

- **deminc** (*bool*) – a logical to indicate if one wants do get rid of the possible azimuthal symmetry
- **bounds** (*bool*) – a boolean to determine if one wants to plot the limits of the domain (True by default)

magic.plotlib.**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[-np.pi:np.pi:gr.nphi*1j, np.pi/2.:-np.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.plotlib.**merContour**(*data*, *radius*, *label=None*, *levels=65*, *cm='seismic'*, *normed=True*, *vmax=None*, *vmin=None*, *cbar=True*, *tit=True*, *fig=None*, *ax=None*, *bounds=True*, *lines=False*)

Plot a meridional cut of a given field

Parameters

- **data** (*numpy.ndarray*) – the input data (an array of size (ntheta,nr))
- **radius** (*numpy.ndarray*) – the input radius
- **label** (*str*) – the name of the input physical quantity you want to display
- **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.
- **bounds** (*bool*) – a boolean to determine if one wants to plot the limits of the domain (True by default)
- **fig** (*matplotlib.figure.Figure*) – a pre-existing figure (if needed)
- **ax** (*matplotlib.axes._subplots.AxesSubplot*) – a pre-existing axis
- **lines** (*bool*) – when set to True, over-plot solid lines to highlight the limits between two adjacent contour levels

magic.plotlib.**radialContour**(*data*, *rad=0.85*, *label=None*, *proj='hammer'*, *lon_0=0.0*, *vmax=None*, *vmin=None*, *lat_0=30.0*, *levels=65*, *cm='seismic'*, *normed=True*, *cbar=True*, *tit=True*, *lines=False*, *fig=None*, *ax=None*)

Plot the radial cut of a given field

Parameters

- **data** (*numpy.ndarray*) – the input data (an array of size (nphi,ntheta))
- **rad** (*float*) – the value of the selected radius
- **label** (*str*) – the name of the input physical quantity 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.
- **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.
- **fig** (*matplotlib.figure.Figure*) – a pre-existing figure (if needed)
- **ax** (*matplotlib.axes._subplots.AxesSubplot*) – a pre-existing axis

9.3.21 Various useful functions

magic.libmagic.**ReadBinaryTimeseries**(*infile*, *ncols*, *datatype='f8'*, *endianness='>'*)

This function reads binary timeseries. It is then faster than the fast_read function.

Parameters

- **infile** (*string*) – the file to read
- **ncols** (*int*) – number of columns of the file
- **datatype** (*string*) – 'f8' = 64-bit floating-point number 'f4' = 32-bit floating-point number
- **endianness** (*string*) – '>' = big-endian ; '<' = small-endian

Returns an array[nlines, ncols] that contains the data of the binary file

Return type numpy.ndarray

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=None*, *std=False*)

This subroutine computes the time-average (and the std) 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
- **std** (*bool*) – when set to True, the standard deviation is also calculated

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 plus one (Nr+1)
- **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.**cylSder**(*radius*, *data*, *order=4*)

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
- **order** (*int*) – order of the finite-difference scheme (possible values are 2 or 4)

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=<class 'numpy.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 (np.float32) or double precision (np.float64)

Returns an array[nlines, ncols] that contains the data of the ascii file

Return type numpy.ndarray

magic.libmagic.**fd_grid**(*nr*, *a*, *b*, *fd_stretching=0.3*, *fd_ratio=0.1*)

This function defines a stretched grid between a and b

```
>>> r_icb = 0.5 ; r_cmb = 1.5; n_r_max=64
>>> rr = fd_grid(n_r_max, r_cmb, r_icb)
```

Parameters

- **nr** (*int*) – number of radial grid points
- **a** (*float*) – upper boundary of the grid
- **b** (*float*) – lower boundary of the grid

- **fd_stretching** (*float*) – fraction of points in the bulk
- **fd_ratio** (*float*) – ratio of minimum to maximum spacing

Returns the radial grid

Returns the radial grid

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.**intcheb**(*f*, *nr*, *z1*, *z2*)
This function integrates an input function f defined on the Gauss-Lobatto grid.

```
>>> print(intcheb(f, 65, 0.5, 1.5))
```

Parameters

- **f** – an input array
- **nr** (*int*) – number of radial grid points
- **z1** (*float*) – lower limit of the Gauss-Lobatto grid
- **z2** (*float*) – upper limit of the Gauss-Lobatto grid

Type numpy.ndarray

Returns the integrated quantity

Return type float

magic.libmagic.**matder**(*nr*, *z1*, *z2*)
This function calculates the derivative in Chebyshev space.

```
>>> r_icb = 0.5 ; r_cmb = 1.5; n_r_max=65
>>> d1 = matder(n_r_max, r_icb, r_cmb)
>>> # Chebyshev grid and data
>>> rr = chebgrid(n_r_max, r_icb, r_cmb)
>>> f = sin(rr)
>>> # Radial derivative
>>> df = dot(d1, f)
```

Parameters

- **nr** (*int*) – number of radial grid points
- **z1** (*float*) – lower limit of the Gauss-Lobatto grid

- **z2** (*float*) – upper limit of the Gauss-Lobatto grid

Returns a matrix of dimension (nr,nr) to calculate the derivatives

Return type numpy.ndarray

magic.libmagic.**phideravg**(*data*, *minc=1*, *order=4*)
phi-derivative of an input array

```
>>> gr = MagicGraph()
>>> dvphidp = phideravg(gr.vphi, minc=gr.minc)
```

Parameters

- **data** (*numpy.ndarray*) – input array
- **minc** (*int*) – azimuthal symmetry
- **order** (*int*) – order of the finite-difference scheme (possible values are 2 or 4)

Returns the phi-derivative of the input array

Return type numpy.ndarray

magic.libmagic.**prime_factors**(*n*)
This function returns all prime factors of a number

Returns all prime factors

Return type list

magic.libmagic.**progressbar**(*it*, *prefix=''*, *size=60*)
Fancy progress-bar for loops

```
for i in progressbar(range(10000000)):
    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.**secondtimeder**(*time*, *y*)

second time derivative of an input array

computed with central differences (numpy.gradient)

```
>>> ts = MagicTs(field='e_kin')
>>> dt_ekinpol = secondtimeder(ts,field='ekin_pol')
```

magic.libmagic.**selectField**(*obj*, *field*, *labTex=True*, *ic=False*)

This function selects for you which field you want to display. It actually allows to avoid possible variables misspelling: i.e. 'Bphi'='bp'='Bp'='bphi'

Parameters

- **obj** (*magic.MagicGraph*) – a graphic output file

- **field** (*str*) – the name of the field one wants to select
- **labTex** (*bool*) – when set to True, format the labels using LaTeX fonts

Returns a tuple that contains the selected physical field and its label

Return type (numpy.ndarray, str)

magic.libmagic.**symmetrize**(*data*, *ms*, *reversed=False*)

Symmetrise an array which is defined only with an azimuthal symmetry minc=ms

Parameters

- **data** (*numpy.ndarray*) – the input array
- **ms** (*int*) – the azimuthal symmetry
- **reversed** (*bool*) – set to True, in case the array is reversed (i.e. n_phi is the last column)

Returns an output array of dimension (data.shape[0]*ms+1)

Return type numpy.ndarray

magic.libmagic.**thetaderavg**(*data*, *order=4*)

Theta-derivative of an input array (finite differences)

```
>>> gr = MagiGraph()
>>> dvtdt = thetaderavg(gr.vtheta)
```

Parameters

- **data** (*numpy.ndarray*) – input array
- **order** (*int*) – order of the finite-difference scheme (possible values are 2 or 4)

Returns the theta-derivative of the input array

Return type numpy.ndarray

magic.libmagic.**timeder**(*time*, *y*)

time derivative of an input array

computed with central differences (numpy.gradient)

```
>>> ts = MagicTs(field='e_kin')
>>> dt_ekinpol = timeder(ts,field='ekin_pol')
```

magic.libmagic.**writeVpEq**(*par*, *tstart*)

This function computes the time-averaged surface zonal flow (and Rolc) and format the output

```
>>> # Reads all the par.* files from the current directory
>>> par = MagicTs(field='par', iplot=False, all=True)
>>> # Time-average
>>> st = writeVpEq(par, tstart=2.1)
>>> print(st)
```

Parameters

- **par** (*magic.MagicTs*) – a *MagicTs* object containing the par file
- **tstart** (*float*) – the starting time of the averaging

Returns a formatted string

Return type str

magic.libmagic.**zderavg**(*data*, *eta=0.35*, *spectral=True*, *colat=None*, *exclude=False*)
z derivative of an input array

```
>>> gr = MagiGraph()
>>> dvrdz = zderavg(gr.vr, eta=gr.radratio, colat=gr.colatitude)
```

Parameters

- **data** (*numpy.ndarray*) – input array
- **eta** (*float*) – aspect ratio of the spherical shell
- **spectral** (*bool*) – when set to True use Chebyshev derivatives, otherwise use finite differences (default is True)
- **exclude** (*bool*) – when set to True, exclude the first and last radial grid points and replace them by a spline extrapolation (default is False)
- **colat** (*numpy.ndarray*) – colatitudes (when not specified a regular grid is assumed)

Returns the z derivative of the input array

Return type numpy.ndarray

CHAPTER

TEN

DESCRIPTION OF THE FORTRAN MODULES

The following pages contain an exhaustive description of the different variables, subroutines and modules used in MagIC. This documentation is automatically generated from the source code docstrings using the Sphinx extention for the Fortran domain.

Fortran modules

1. For the main program file `magic.f90`, see *here*.
2. For the core modules that contain most of the global variables, see *here*.
3. For the MPI related modules, see *here*.
4. For the code initialization and the pre-calculations done in the initial stage of the computation (before the time-stepping loop), see *here* and *there*.
5. For the time-stepping loop, see *here*.
6. For the calculation of the non-linear terms (in the physical space) and their time-advance, see *here*.
7. For the calculation of the linear terms (in spectral space) and their time-advance, see *here*.
8. For the Chebyshev, Fourier and Legendre transforms, see *here*.
9. For the computation of the radial derivatives (Chebyshev) and the integration, see *here*.
10. For the definition of the blocking, see *here*.
11. For the calculation of the standard outputs (time-series, spectra and radial files), see *here*.
12. For the calculation of binary outputs (graphic files, movie files, potential and coeff files), see *here*.
13. For the additional calculations of specific outputs (torsional oscillations, RMS force balance, etc.), see *here*.
14. For reading and writing the check points (restart files), see *here*.
15. For additional useful functions (string manipulation, 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 iso_fortran_env, *charmanip* (*write_long_string()*), *truncation*, *precision_mod*, *physical_parameters*, *courant_mod* (*initialize_courant()*, *finalize_courant()*), *radial_der* (*initialize_der_arrays()*, *finalize_der_arrays()*), *radial_functions* (*initialize_radial_functions()*, *finalize_radial_functions()*), *num_param*, *torsional_oscillations*, *init_fields*, *special* (*initialize_grenoble()*, *finalize_grenoble()*), *blocking* (*initialize_blocking()*, *finalize_blocking()*, *llm()*, *ulm()*), *timing* (timer_type()), *horizontal_data*, *logic*, *fields*, *fieldslast*, *constants* (*codeversion()*), *movie_data* (*initialize_movie_data()*, *finalize_movie_data()*), *rms* (*initialize_rms()*, *finalize_rms()*), *dtb_mod* (*initialize_dtb_mod()*, *finalize_dtb_mod()*), *radial_data* (*initialize_radial_data()*, *finalize_radial_data()*), *radialloop* (*initialize_radialloop()*, *finalize_radialloop()*), *lmloop_mod* (*initialize_lmloop()*, *finalize_lmloop()*, *test_lmloop()*), *precalculations*, *start_fields* (*getstartfields()*), *kinetic_energy*, *magnetic_energy*, *fields_average_mod*, *geos* (*initialize_geos()*, *finalize_geos()*), *spectra* (*initialize_spectra()*, *finalize_spectra()*), *output_data* (*tag()*, *log_file()*, *n_log_file()*), *output_mod* (*initialize_output()*, *finalize_output()*), *outto_mod* (*initialize_outto_mod()*, *finalize_outto_mod()*), *parallel_mod*, *namelists*, *step_time_mod* (*initialize_step_time()*, *step_time()*), *communications* (*initialize_communications()*, *finalize_communications()*), *power* (*initialize_output_power()*, *finalize_output_power()*), *outpar_mod* (*initialize_outpar_mod()*, *finalize_outpar_mod()*), *outmisc_mod* (*initialize_outmisc_mod()*, *finalize_outmisc_mod()*), *outrot* (*initialize_outrot()*, *finalize_outrot()*), *mem_alloc*, *useful* (*abortrun()*), *probe_mod* (*initialize_probes()*, *finalize_probes()*), *time_schemes* (type_tscheme()), *sht* (*initialize_sht()*, *finalize_sht()*)

Call to *parallel()*, *write_long_string()*, *readnamelists()*, *initialize_output()*, *checktruncation()*, *initialize_memory_counter()*, *initialize_blocking()*, *initialize_sht()*, *initialize_radial_data()*, *initialize_radial_functions()*, *initialize_horizontal_data()*, *memwrite()*, *initialize_radialloop()*, *initialize_lmloop()*, *initialize_num_param()*, *initialize_init_fields()*, *initialize_grenoble()*, *initialize_fields()*, *initialize_fieldslast()*, *initialize_step_time()*, *initialize_communications()*, *initialize_der_arrays()*, *initialize_kinetic_energy()*, *initialize_magnetic_energy()*, *initialize_spectra()*, *initialize_outpar_mod()*, *initialize_outmisc_mod()*, *initialize_outrot()*, *initialize_output_power()*, *initialize_fields_average_mod()*, *initialize_to()*, *precalc()*, *initialize_geos()*, *initialize_outto_mod()*, *initialize_movie_data()*, *initialize_dtb_mod()*, *initialize_probes()*, *initialize_rms()*, *finalize_memory_counter()*, *writenamelists()*, *getstartfields()*, *initialize_courant()*, *precalctimes()*, *writeinfo()*, *test_lmloop()*, *step_time()*, *finalize_movie_data()*, *finalize_rms()*, *finalize_outto_mod()*, *finalize_to()*, *finalize_geos()*, *finalize_dtb_mod()*, *finalize_fields_average_mod()*, *finalize_output_power()*, *finalize_outrot()*, *finalize_outmisc_mod()*, *finalize_outpar_mod()*, *finalize_spectra()*, *finalize_magnetic_energy()*, *finalize_kinetic_energy()*, *finalize_probes()*,

finalize_courant(), *finalize_communications()*, *finalize_fieldslast()*, *finalize_fields()*, *finalize_grenoble()*, *finalize_init_fields()*, *finalize_num_param()*, *finalize_lmloop()*, *finalize_radialloop()*, *finalize_sht()*, *finalize_der_arrays()*, *finalize_horizontal_data()*, *finalize_radial_functions()*, *finalize_blocking()*, *finalize_radial_data()*, *finalize_output()*

10.2 Base modules

10.2.1 precision.f90

Description

This module controls the precision used in MagIC

Quick access

Variables *cp*, *lip*, *mpi_def_complex*, *mpi_def_real*, *mpi_out_real*, *outp*, *sizeof_character*, *sizeof_def_complex*, *sizeof_def_real*, *sizeof_integer*, *sizeof_logical*, *sizeof_out_real*

Needed modules

- `iso_fortran_env` (`real32()`, `real64()`, `int32()`, `int64()`)
- `mpimod`

Variables

- `precision_mod/`**`cp`** *[integer,parameter*=real64*]*
- `precision_mod/`**`lip`** *[integer,parameter*=int64*]*
- `precision_mod/`**`mpi_def_complex`** *[integer,parameter*=mpi_complex16*]*
- `precision_mod/`**`mpi_def_real`** *[integer,parameter*=mpi_real8*]*
- `precision_mod/`**`mpi_out_real`** *[integer,parameter*=mpi_real4*]*
- `precision_mod/`**`outp`** *[integer,parameter*=real32*]*
- `precision_mod/`**`sizeof_character`** *[integer,parameter*=1*]*
- `precision_mod/`**`sizeof_def_complex`** *[integer,parameter*='real64real64'*]*
- `precision_mod/`**`sizeof_def_real`** *[integer,parameter*='real64'*]*
- `precision_mod/`**`sizeof_integer`** *[integer,parameter*=int32*]*
- `precision_mod/`**`sizeof_logical`** *[integer,parameter*=2*]*
- `precision_mod/`**`sizeof_out_real`** *[integer,parameter*='real32'*]*

10.2.2 truncation.f90

Description

This module defines the grid points and the truncation

Quick access

Variables *fd_order*, *fd_order_bound*, *fd_ratio*, *fd_stretch*, *l_axi*, *l_max*, *l_maxmag*, *ldtbmem*, *lm_max*, *lm_max_dtb*, *lm_maxmag*, *lmagmem*, *lmoviemem*, *lmp_max*, *lmp_max_dtb*, *lstressmem*, *m_max*, *minc*, *n_cheb_ic_max*, *n_cheb_max*, *n_m_max*, *n_phi_max*, *n_phi_maxstr*, *n_phi_tot*, *n_r_ic_max*, *n_r_ic_max_dtb*, *n_r_ic_maxmag*, *n_r_max*, *n_r_max_dtb*, *n_r_maxmag*, *n_r_maxstr*, *n_r_tot*, *n_r_totmag*, *n_theta_axi*, *n_theta_max*, *n_theta_maxstr*, *nalias*, *nlat_padded*, *radial_scheme*

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

Needed modules

- *precision_mod* (*cp()*): This module controls the precision used in MagIC
- *logic* (*l_finite_diff()*, *l_cond_ic()*): Module containing the logicals that control the run
- *useful* (*abortrun()*): This module contains several useful routines.

Variables

- truncation/**fd_order** *[integer]*
  Finite difference order (for now only 2 and 4 are safe)
- truncation/**fd_order_bound** *[integer]*
  Finite difference order on the boundaries
- truncation/**fd_ratio** *[real]*
  drMin over drMax (only when FD are used)
- truncation/**fd_stretch** *[real]*
  regular intervals over irregular intervals
- truncation/**l_axi** *[logical]*
  logical for axisymmetric calculations
- truncation/**l_max** *[integer]*
  max degree of Plms
- truncation/**l_maxmag** *[integer]*
  Max. degree for magnetic field calculation
- truncation/**ldtbmem** *[integer]*
  Memory for movie output
- truncation/**lm_max** *[integer]*
  number of l/m combinations
- truncation/**lm_max_dtb** *[integer]*
  Number of l/m combinations for movie output

- truncation/**lm_maxmag** *[integer]*
  Max. number of l/m combinations for magnetic field calculation
- truncation/**lmagmem** *[integer]*
  Memory for magnetic field calculation
- truncation/**lmoviemem** *[integer]*
  Memory for movies
- truncation/**lmp_max** *[integer]*
  number of l/m combination if l runs to l_max+1
- truncation/**lmp_max_dtb** *[integer]*
  Number of l/m combinations for movie output if l runs to l_max+1
- truncation/**lstressmem** *[integer]*
  Memory for stress output
- truncation/**m_max** *[integer]*
  max order of Plms
- truncation/**minc** *[integer]*
  basic wavenumber, longitude symmetry
- truncation/**n_cheb_ic_max** *[integer]*
  number of chebs in inner core
- truncation/**n_cheb_max** *[integer]*
  max degree-1 of cheb polynomia
- truncation/**n_m_max** *[integer]*
  max number of ms (different oders)
- truncation/**n_phi_max** *[integer]*
  absolute number of phi grid-points
- truncation/**n_phi_maxstr** *[integer]*
  Number of phi points for stress output
- truncation/**n_phi_tot** *[integer]*
  number of longitude grid points
- truncation/**n_r_ic_max** *[integer]*
  number of grid points in inner core
- truncation/**n_r_ic_max_dtb** *[integer]*
  Number of IC radial points for movie output
- truncation/**n_r_ic_maxmag** *[integer]*
  Number of radial points to calculate IC magnetic field
- truncation/**n_r_max** *[integer]*
  number of radial grid points
- truncation/**n_r_max_dtb** *[integer]*
  Number of radial points for movie output
- truncation/**n_r_maxmag** *[integer]*
  Number of radial points to calculate magnetic field
- truncation/**n_r_maxstr** *[integer]*
  Number of radial points for stress output

- truncation/**n_r_tot** *[integer]*
  total number of radial grid points
- truncation/**n_r_totmag** *[integer]*
  n_r_maxMag + n_r_ic_maxMag
- truncation/**n_theta_axi** *[integer]*
  number of theta grid-points (axisymmetric models)
- truncation/**n_theta_max** *[integer]*
  number of theta grid-points
- truncation/**n_theta_maxstr** *[integer]*
  Number of theta points for stress output
- truncation/**nalias** *[integer]*
  controls dealiasing in latitude
- truncation/**nlat_padded** *[integer]*
  number of theta grid-points with padding included
- truncation/**radial_scheme** *[character]*
  radial scheme (either Cheybev of FD)

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*

Call to *abortrun()*

10.2.3 num_param.f90

Description

Module containing numerical and control parameters

Quick access

Variables *alffac*, *alph1*, *alph2*, *alpha*, *amstart*, *anelastic_flavour*, *courfac*, *dct_counter*, *delxh2*, *delxr2*, *difchem*, *difeta*, *difkap*, *difnu*, *dtmax*, *dtmin*, *enscale*, *escale*, *f_exp_counter*, *intfac*, *istop*, *ldif*, *ldifexp*, *lm2phy_counter*, *lscale*, *map_function*, *mpi_packing*, *mpi_transp*, *n_lscale*, *n_time_steps*, *n_tscale*, *nl_counter*, *phy2lm_counter*, *polo_flow_eq*, *pscale*, *run_time_limit*, *solve_counter*, *td_counter*, *tend*, *time_scheme*, *timestart*, *tscale*, *vscale*

Routines *finalize_num_param()*, *initialize_num_param()*

Needed modules

- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *truncation* (*n_r_max()*): This module defines the grid points and the truncation
- *timing* (timer_type()): This module contains functions that are used to measure the time spent.
- *precision_mod*: This module controls the precision used in MagIC

Variables

- num_param/**alffac** *[real,public]*
  Value to scale Alfen-velocity in courant criteria
- num_param/**alph1** *[real,public]*
  Input parameter for non-linear map to define degree of spacing (0.0:2.0)
- num_param/**alph2** *[real,public]*
  Input parameter for non-linear map to define central point of different spacing (-1.0:1.0)
- num_param/**alpha** *[real,public]*
  Weight for implicit time step
- num_param/**amstart** *[real,public]*
- num_param/**anelastic_flavour** *[character,public]*
  version of the anelastic approximation
- num_param/**courfac** *[real,public]*
  Value to scale velocity in courant criteria
- num_param/**dct_counter** *[timer_type,public]*
  Time counter for discrete cosine transforms
- num_param/**delxh2** (*) *[real,allocatable/public]*
  Auxiliary arrays containing effective Courant grid intervals
- num_param/**delxr2** (*) *[real,allocatable/public]*
  Auxiliary arrays containing effective Courant grid intervals
- num_param/**difchem** *[real,public]*
  Amplitude of chemical hyperdiffusion
- num_param/**difeta** *[real,public]*
  Amplitude of magnetic hyperdiffusion
- num_param/**difkap** *[real,public]*
  Amplitude of thermal hyperdiffusion
- num_param/**difnu** *[real,public]*
  Amplitude of viscous hyperdiffusion
- num_param/**dtmax** *[real,public]*
  Maximum allowed time step
- num_param/**dtmin** *[real,public]*
  Minimum allowed time step
- num_param/**enscale** *[real,public]*
  Energies scale

- num_param/**escale** *[real,public]*
  Energy scale
- num_param/**f_exp_counter** *[timer_type,public]*
  Time counter for r-der of adv. terms
- num_param/**intfac** *[real,public]*
  Value to re-scale dtMax during simulation
- num_param/**istop** *[integer,public]*
  Variable used in FFT soubroutine
- num_param/**ldif** *[integer,public]*
  Degree where hyperdiffusion starts to act
- num_param/**ldifexp** *[integer,public]*
  Exponent for hyperdiffusion function
- num_param/**lm2phy_counter** *[timer_type,public]*
- num_param/**lscale** *[real,public]*
  Length scale
- num_param/**map_function** *[character,public]*
  Mapping family: either tangent or arcsin
- num_param/**mpi_packing** *[character,public]*
  packing the alltoall
- num_param/**mpi_transp** *[character,public]*
  Form of the MPI transpose (point to point or alltoall)
- num_param/**n_lscale** *[integer,public]*
  Control length scale
- num_param/**n_time_steps** *[integer,public]*
  Total number of time steps requested in the name list
- num_param/**n_tscale** *[integer,public]*
  Control time scale
- num_param/**nl_counter** *[timer_type,public]*
- num_param/**phy2lm_counter** *[timer_type,public]*
- num_param/**polo_flow_eq** *[character,public]*
  form of the poloidal flow equation: Pressure or Double Curl
- num_param/**pscale** *[real,public]*
- num_param/**run_time_limit** *[real,public]*
- num_param/**solve_counter** *[timer_type,public]*
  Time counter for linear solves
- num_param/**td_counter** *[timer_type,public]*
- num_param/**tend** *[real,public]*
  Numerical time where run should end
- num_param/**time_scheme** *[character,public]*
  Time scheme
- num_param/**timestart** *[real,public]*
  Numerical time where run should start

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

Subroutines and functions

subroutine num_param/**initialize_num_param**()

Called from *magic*

subroutine num_param/**finalize_num_param**()

Called from *magic*

10.2.4 phys_param.f90

Description

Module containing the physical parameters

Quick access

Variables *ampstrat*, *bn*, *buofac*, *chemfac*, *cmbhflux*, *con_decrate*, *con_funcwidth*, *con_lambdamatch*, *con_lambdaout*, *con_radratio*, *conductance_ma*, *corfac*, *difexp*, *dilution_fac*, *dissnb*, *ek*, *ekscaled*, *epsc*, *epsc0*, *epscxi*, *epscxi0*, *epsphase*, *epss*, *g0*, *g1*, *g2*, *grunnb*, *imagcon*, *imps*, *impxi*, *interior_model*, *kbotb*, *kbotphi*, *kbots*, *kbotv*, *kbotxi*, *ktopb*, *ktopp*, *ktopphi*, *ktops*, *ktopv*, *ktopxi*, *lffac*, *mode*, *n_imps*, *n_imps_max*, *n_impxi*, *n_impxi_max*, *n_r_lcr*, *nvarcond*, *nvardiff*, *nvarentropygrad*, *nvareps*, *nvarvisc*, *o_sr*, *oek*, *ohmlossfac*, *opm*, *opr*, *osc*, *peaks*, *peakxi*, *penaltyfac*, *phasedifffac*, *phis*, *phixi*, *po*, *polind*, *pr*, *prec_angle*, *prmag*, *r_cut_model*, *r_lcr*, *ra*, *radratio*, *rascaled*, *raxi*, *rho_ratio_ic*, *rho_ratio_ma*, *rstrat*, *sc*, *sigma_ratio*, *slopestrat*, *stef*, *strat*, *thetas*, *thetaxi*, *thexpnb*, *thickstrat*, *tmagcon*, *tmelt*, *vischeatfac*, *widths*, *widthxi*

Needed modules

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

Variables

- physical_parameters/**ampstrat** *[real]*
  stratified Layer
- physical_parameters/**bn** *[real]*
  Normalisation of He burning
- physical_parameters/**buofac** *[real]*
  Ratio of Rayleigh number over Prandtl number

- physical_parameters/**chemfac** *[real]*
  Ratio of comp. Rayleigh number over Schmidt number
- physical_parameters/**cmbhflux** *[real]*
  stratified Layer
- physical_parameters/**con_decrate** *[real]*
  Slope of electrical conductivity profile (nVarCond=2)
- physical_parameters/**con_funcwidth** *[real]*
  nVarCond=1
- physical_parameters/**con_lambdamatch** *[real]*
  Electrical conductivity at con_RadRatio (nVarCond=2)
- physical_parameters/**con_lambdaout** *[real]*
  nVarCond=1
- physical_parameters/**con_radratio** *[real]*
  Transition between branches of electrical conductivity profile (nVarCond=1,2)
- physical_parameters/**conductance_ma** *[real]*
  OC conductivity
- physical_parameters/**corfac** *[real]*
  Inverse of ekScaled
- physical_parameters/**difexp** *[real]*
  Thermal diffusivity variation
- physical_parameters/**dilution_fac** *[real]*
  Omega^2 d/g_top for centrifugal acceleration, named after Chandrasekhar (1987)
- physical_parameters/**dissnb** *[real]*
  Dissipation number
- physical_parameters/**ek** *[real]*
  Ekman number
- physical_parameters/**ekscaled** *[real]*
  $E\,l^2$
- physical_parameters/**epsc** *[real]*
  Renormalisation of epsc0
- physical_parameters/**epsc0** *[real]*
  Internal heat source magnitude
- physical_parameters/**epscxi** *[real]*
  Renormalisation of epsc0Xi
- physical_parameters/**epscxi0** *[real]*
  Internal chemical heat source magnitude
- physical_parameters/**epsphase** *[real]*
  Cahn number for phase field equatioo
- physical_parameters/**epss** *[real]*
  Deviation from the adiabat
- physical_parameters/**g0** *[real]*
  Set to 1.0 for constant gravity

- physical_parameters/**g1** *[real]*
  Set to 1.0 for linear gravity
- physical_parameters/**g2** *[real]*
  Set to 1.0 for $1/r^2$ gravity
- physical_parameters/**grunnb** *[real]*
  Grüneisen paramater $\Gamma = (\gamma - 1)/\alpha T$
- physical_parameters/**imagcon** *[integer]*
  Imposed magnetic field for magnetoconvection, at the boundaries
- physical_parameters/**imps** *[integer]*
  Heat boundary condition
- physical_parameters/**impxi** *[integer]*
- physical_parameters/**interior_model** *[character]*
  name of the interior model
- physical_parameters/**kbotb** *[integer]*
  Magnetic boundary condition
- physical_parameters/**kbotphi** *[integer]*
  Boundary conditions for phase field
- physical_parameters/**kbots** *[integer]*
  Entropy boundary condition
- physical_parameters/**kbotv** *[integer]*
  Velocity boundary condition
- physical_parameters/**kbotxi** *[integer]*
  Boundary conditions for chemical composition
- physical_parameters/**ktopb** *[integer]*
- physical_parameters/**ktopp** *[integer]*
  Boundary condition for spherically-symmetric pressure
- physical_parameters/**ktopphi** *[integer]*
- physical_parameters/**ktops** *[integer]*
- physical_parameters/**ktopv** *[integer]*
- physical_parameters/**ktopxi** *[integer]*
- physical_parameters/**lffac** *[real]*
  Inverse of Pr*Ekman
- physical_parameters/**mode** *[integer]*
  Mode of calculation
- physical_parameters/**n_imps** *[integer]*
  Heat boundary condition
- physical_parameters/**n_imps_max** *[integer,parameter=20]*
  Heat boundary condition
- physical_parameters/**n_impxi** *[integer]*
- physical_parameters/**n_impxi_max** *[integer,parameter=20]*
- physical_parameters/**n_r_lcr** *[integer]*
  Number of radial points where conductivity is zero

- physical_parameters/**nvarcond** *[integer]*
  Selection of variable conductivity profile
- physical_parameters/**nvardiff** *[integer]*
  Selection of variable diffusivity profile
- physical_parameters/**nvarentropygrad** *[integer]*
  stratified Layer
- physical_parameters/**nvareps** *[integer]*
  Selection of internal heating profile
- physical_parameters/**nvarvisc** *[integer]*
  Selection of variable viscosity profile
- physical_parameters/**o_sr** *[real]*
  Inverse of sigma_ratio
- physical_parameters/**oek** *[real]*
  Inverse of the Ekman number
- physical_parameters/**ohmlossfac** *[real]*
  Prefactor for Ohmic heating: $Di\,Pr/(Ra\,E\,Pm^2)$
- physical_parameters/**opm** *[real]*
  Inverse of magnetic Prandtl number
- physical_parameters/**opr** *[real]*
  Inverse of Prandtl number
- physical_parameters/**osc** *[real]*
  Inverse of Schmidt number (i.e. chemical Prandtl number)
- physical_parameters/**peaks** (20) *[real]*
- physical_parameters/**peakxi** (20) *[real]*
- physical_parameters/**penaltyfac** *[real]*
  Factor that enters the penalty method in the NS equations
- physical_parameters/**phasedifffac** *[real]*
  Diffusion term of phase field
- physical_parameters/**phis** (20) *[real]*
- physical_parameters/**phixi** (20) *[real]*
- physical_parameters/**po** *[real]*
  Poincaré number
- physical_parameters/**polind** *[real]*
  polytropic index
- physical_parameters/**pr** *[real]*
  Prandtl number
- physical_parameters/**prec_angle** *[real]*
  Precession angle
- physical_parameters/**prmag** *[real]*
  magnetic Prandtl number
- physical_parameters/**r_cut_model** *[real]*
  Percentage on the inner part of the interior model to be used

- physical_parameters/**r_lcr** *[real]*
  Radius beyond which conductivity is zero
- physical_parameters/**ra** *[real]*
  Rayleigh number
- physical_parameters/**radratio** *[real]*
  aspect ratio
- physical_parameters/**rascaled** *[real]*
  $Ra\, l^3$
- physical_parameters/**raxi** *[real]*
  Chemical composition-based Rayleigh number
- physical_parameters/**rho_ratio_ic** *[real]*
  Same density as outer core
- physical_parameters/**rho_ratio_ma** *[real]*
  Same density as outer core
- physical_parameters/**rstrat** *[real]*
  stratified Layer
- physical_parameters/**sc** *[real]*
  Schmidt number (i.e. chemical Prandtl number)
- physical_parameters/**sigma_ratio** *[real]*
  Value of IC rotation
- physical_parameters/**slopestrat** *[real]*
  stratified Layer
- physical_parameters/**stef** *[real]*
  Stefan number
- physical_parameters/**strat** *[real]*
  number of density scale heights
- physical_parameters/**thetas** (20) *[real]*
- physical_parameters/**thetaxi** (20) *[real]*
- physical_parameters/**thexpnb** *[real]*
  Thermal expansion * temperature $\alpha_0 T_0$
- physical_parameters/**thickstrat** *[real]*
  stratified Layer
- physical_parameters/**tmagcon** *[real]*
  Time for magnetoconvection calculation
- physical_parameters/**tmelt** *[real]*
  Melting temperature
- physical_parameters/**vischeatfac** *[real]*
  Prefactor for viscous heating: $Di\, Pr/Ra$
- physical_parameters/**widths** (20) *[real]*
- physical_parameters/**widthxi** (20) *[real]*

10.2.5 logic.f90

Description

Module containing the logicals that control the run

Quick access

Variables *l_2d_rms*, *l_2d_spectra*, *l_ab1*, *l_adv_curl*, *l_am*, *l_anel*, *l_anelastic_liquid*, *l_average*, *l_b_nl_cmb*, *l_b_nl_icb*, *l_bridge_step*, *l_centrifuge*, *l_chemical_conv*, *l_cmb_field*, *l_cond_ic*, *l_cond_ma*, *l_conv*, *l_conv_nl*, *l_corr*, *l_correct_ame*, *l_correct_amz*, *l_corrmov*, *l_cour_alf_damp*, *l_double_curl*, *l_drift*, *l_dt_cmb_field*, *l_dtb*, *l_dtbmovie*, *l_dtrmagspec*, *l_earth_likeness*, *l_energy_modes*, *l_finite_diff*, *l_fluxprofs*, *l_full_sphere*, *l_heat*, *l_heat_nl*, *l_hel*, *l_ht*, *l_htmovie*, *l_iner*, *l_isothermal*, *l_lcr*, *l_mag*, *l_mag_kin*, *l_mag_lf*, *l_mag_nl*, *l_mag_par_solve*, *l_movie*, *l_movie_ic*, *l_movie_oc*, *l_newmap*, *l_non_adia*, *l_non_rot*, *l_packed_transp*, *l_par*, *l_parallel_solve*, *l_perppar*, *l_phase_field*, *l_power*, *l_precession*, *l_pressgraph*, *l_probe*, *l_r_field*, *l_r_fieldt*, *l_r_fieldxi*, *l_rmagspec*, *l_rms*, *l_rot_ic*, *l_rot_ma*, *l_runtimelimit*, *l_save_out*, *l_single_matrix*, *l_spec_avg*, *l_sric*, *l_srma*, *l_store_frame*, *l_temperature_diff*, *l_to*, *l_tomovie*, *l_update_b*, *l_update_s*, *l_update_v*, *l_update_xi*, *l_var_l*, *l_viscbccalc*, *l_z10mat*, *lverbose*

Variables

- logic/**l_2d_rms** *[logical]*
  Switch for storing of time-averaged r-l-spectra of forces
- logic/**l_2d_spectra** *[logical]*
  Switch for storing of r-l-spectra
- logic/**l_ab1** *[logical]*
  1st order Adams Bashforth
- logic/**l_adv_curl** *[logical]*
  Use curl{u}times u for the advection
- logic/**l_am** *[logical]*
  Switch for angular momentum calculation
- logic/**l_anel** *[logical]*
  Switch for anelastic calculation
- logic/**l_anelastic_liquid** *[logical]*
  Switch for anelastic liquid calculation
- logic/**l_average** *[logical]*
  Switch for calculation of time-averages
- logic/**l_b_nl_cmb** *[logical]*
  Switch for non-linear magnetic field at OC
- logic/**l_b_nl_icb** *[logical]*
  Switch for non-linear magnetic field at IC
- logic/**l_bridge_step** *[logical]*
  Used to bridge missing steps when changing the time integrator

- logic/**l_centrifuge** *[logical]*
  Compute centrifugal acceleration
- logic/**l_chemical_conv** *[logical]*
  Switch for chemical convection
- logic/**l_cmb_field** *[logical]*
  Switch for Bcoef files for gauss coefficients
- logic/**l_cond_ic** *[logical]*
  Switch for conducting IC
- logic/**l_cond_ma** *[logical]*
  Switch for conducting OC
- logic/**l_conv** *[logical]*
  Switch off convection
- logic/**l_conv_nl** *[logical]*
  Switch off non-linear convection terms
- logic/**l_corr** *[logical]*
  Switch off rotation
- logic/**l_correct_ame** *[logical]*
  Switch for correction of equatorial angular mom.
- logic/**l_correct_amz** *[logical]*
  Switch for correction of axial angular momentum
- logic/**l_corrmov** *[logical]*
  Switch for North/south correlation movie (see s_getEgeos.f)
- logic/**l_cour_alf_damp** *[logical]*
  Modified Alfven Courant condition based on Christensen et al., GJI, 1999 (.true. by default)
- logic/**l_double_curl** *[logical]*
  Use the double-curl of the NS equation to get the poloidal equation
- logic/**l_drift** *[logical]*
  Switch for drift rates calculation
- logic/**l_dt_cmb_field** *[logical]*
  Switch for Bcoef files for secular variation of gauss coefs.
- logic/**l_dtb** *[logical]*
  Switch to reserve memory for dtB movie
- 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_earth_likeness** *[logical]*
  Compute the Earth-likeness of the CMB field following Christensen et al., EPSL, 2010
- logic/**l_energy_modes** *[logical]*
  Switch for calculation of distribution of energies over m's
- logic/**l_finite_diff** *[logical]*
  Use finite differences for the radial scheme

- logic/**l_fluxprofs** *[logical]*
  Switch for calculation of radial profiles of flux contributions
- logic/**l_full_sphere** *[logical]*
  Set to .true. if this is a full sphere calculation
- logic/**l_heat** *[logical]*
  Switch off heat terms calculation
- logic/**l_heat_nl** *[logical]*
  Switch off non-linear heat terms calculation
- logic/**l_hel** *[logical]*
  Switch for helicity calculation, output in misc.TAG
- logic/**l_ht** *[logical]*
  Switch for heat flux movie frame output
- logic/**l_htmovie** *[logical]*
  Switch for heat flux movie output
- logic/**l_iner** *[logical]*
  Switch for inertial modes calculation
- logic/**l_isothermal** *[logical]*
  Switch for isothermal calculation
- logic/**l_lcr** *[logical]*
  Switch for zero electrical conductivity beyond r_LCR
- logic/**l_mag** *[logical]*
  Switch off magnetic terms calculation
- logic/**l_mag_kin** *[logical]*
  Switch related for kinematic dynamo
- logic/**l_mag_lf** *[logical]*
  Switch off Lorentz force term
- logic/**l_mag_nl** *[logical]*
  Switch off non-linear magnetic terms calculation
- logic/**l_mag_par_solve** *[logical]*
  Can be remove once inner core has also been ported
- logic/**l_movie** *[logical]*
  Switch for recording of movie files
- logic/**l_movie_ic** *[logical]*
  Switch for recording of movie files for IC
- logic/**l_movie_oc** *[logical]*
  Switch for recording of movie files for OC
- logic/**l_newmap** *[logical]*
  Switch for non-linear mapping (see Bayliss and Turkel, 1990)
- logic/**l_non_adia** *[logical]*
  Switch in case the reference state is non-adiabatic
- logic/**l_non_rot** *[logical]*
  Switch to non-rotating

- logic/**l_packed_transp** *[logical]*
  Pack or don't pack MPI transposes
- logic/**l_par** *[logical]*
  Switch for additional parameters calculation in s_getEgeos.f
- logic/**l_parallel_solve** *[logical]*
  Use R-distributed parallel solver (work only for F.D.)
- logic/**l_perppar** *[logical]*
  Switch for calculation of of kinetic energy perpendicular+parallel to the rotation axis
- logic/**l_phase_field** *[logical]*
  Switch when phase field is used
- logic/**l_power** *[logical]*
  Switch for power budget terms calculation
- logic/**l_precession** *[logical]*
  Use precession
- logic/**l_pressgraph** *[logical]*
  Store pressure in graphic files
- logic/**l_probe** *[logical]*
  Switch for artifical sensors
- logic/**l_r_field** *[logical]*
  Switch for radial coefficients
- logic/**l_r_fieldt** *[logical]*
  Switch for radial T coefficients
- logic/**l_r_fieldxi** *[logical]*
  Switch for radial Xi coefficients
- logic/**l_rmagspec** *[logical]*
  Switch for magnetic spectra at different depths at log times
- logic/**l_rms** *[logical]*
  Switch for RMS force balances calculation
- logic/**l_rot_ic** *[logical]*
  Switch off IC rotation
- logic/**l_rot_ma** *[logical]*
  Switch off OC rotation
- logic/**l_runtimelimit** *[logical]*
  Switch for absolute time limit of the run
- logic/**l_save_out** *[logical]*
  Switch off outputs
- logic/**l_single_matrix** *[logical]*
  In case entropy, w and P are solved at once implicitely
- logic/**l_spec_avg** *[logical]*
  Switch for calculation of time-averaged spectra
- logic/**l_sric** *[logical]*
  Switch to rotating IC with prescribed rot. rate

- logic/**l_srma** *[logical]*
  Switch to rotating OC with prescribed rot. rate
- logic/**l_store_frame** *[logical]*
  Switch for storing movie frames
- logic/**l_temperature_diff** *[logical]*
  diffusion of temperature instead of entropy
- logic/**l_to** *[logical]*
  Switch for TO output in TOnhs.TAG, TOshs.TAG
- logic/**l_tomovie** *[logical]*
  Switch for TO movie output
- logic/**l_update_b** *[logical]*
  Switch off magnetic field update
- logic/**l_update_s** *[logical]*
  Switch off entropy update
- logic/**l_update_v** *[logical]*
  Switch off velocity field update
- logic/**l_update_xi** *[logical]*
  Switch off update of chemical composition
- logic/**l_var_l** *[logical]*
  When set to .true., degree varies with radius
- logic/**l_viscbccalc** *[logical]*
  Switch for dissipation layer for stress-free BCs plots
- logic/**l_z10mat** *[logical]*
  Switch for solid body rotation
- logic/**lverbose** *[logical]*
  Switch for detailed information about run progress

10.2.6 fields.f90

Description

This module contains all the fields used in MagIC in the hybrid (LM,r) space as well as their radial derivatives. It defines both the LM-distributed arrays and the R-distributed arrays.

Quick access

Variables *aj_ic*, *aj_ic_lmloc*, *aj_lmloc*, *aj_rloc*, *b_ic*, *b_ic_lmloc*, *b_lmloc*, *b_rloc*, *db_ic*, *db_ic_lmloc*, *db_lmloc*, *db_rloc*, *ddb_ic*, *ddb_ic_lmloc*, *ddb_lmloc*, *ddb_rloc*, *ddj_ic*, *ddj_ic_lmloc*, *ddj_lmloc*, *ddj_rloc*, *ddw_lmloc*, *ddw_rloc*, *dj_ic*, *dj_ic_lmloc*, *dj_lmloc*, *dj_rloc*, *dp_lmloc*, *dp_rloc*, *ds_lmloc*, *ds_rloc*, *dw_lmloc*, *dw_rloc*, *dxi_lmloc*, *dxi_rloc*, *dz_lmloc*, *dz_rloc*, *field_lmloc_container*, *field_rloc_container*, *flow_lmloc_container*, *flow_rloc_container*, *omega_ic*, *omega_ma*, *p_lmloc*, *p_rloc*, *phi_lmloc*, *phi_rloc*, *press_lmloc_container*, *press_rloc_container*, *s_lmloc*, *s_lmloc_container*, *s_rloc*, *s_rloc_container*, *w_lmloc*, *w_rloc*, *work_lmloc*, *xi_lmloc*, *xi_lmloc_container*, *xi_rloc*, *xi_rloc_container*, *z_lmloc*, *z_rloc*

Routines *finalize_fields()*, *initialize_fields()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *truncation* (*lm_max()*, *n_r_max()*, *lm_maxmag()*, *n_r_maxmag()*, *n_r_ic_maxmag()*, *fd_order()*, *fd_order_bound()*): This module defines the grid points and the truncation
- *logic* (*l_chemical_conv()*, *l_finite_diff()*, *l_mag()*, *l_parallel_solve()*, *l_mag_par_solve()*, *l_phase_field()*): Module containing the logicals that control the run
- *blocking* (*llm()*, *ulm()*, *llmmag()*, *ulmmag()*): Module containing blocking information
- *radial_data* (*nrstart()*, *nrstop()*, *nrstartmag()*, *nrstopmag()*): This module defines the MPI decomposition in the radial direction.
- *parallel_mod* (`rank()`): This module contains the blocking information

Variables

- `fields/`**`aj_ic`** (*,*) *[complex,allocatable/public]*
- `fields/`**`aj_ic_lmloc`** (*,*) *[complex,allocatable/public]*
- `fields/`**`aj_lmloc`** (*,*) *[complex,pointer/public]*
- `fields/`**`aj_rloc`** (*,*) *[complex,pointer/public]*
- `fields/`**`b_ic`** (*,*) *[complex,allocatable/public]*
- `fields/`**`b_ic_lmloc`** (*,*) *[complex,allocatable/public]*
- `fields/`**`b_lmloc`** (*,*) *[complex,pointer/public]*
- `fields/`**`b_rloc`** (*,*) *[complex,pointer/public]*
- `fields/`**`bicb`** (*) *[complex,allocatable/public]*
- `fields/`**`db_ic`** (*,*) *[complex,allocatable/public]*
- `fields/`**`db_ic_lmloc`** (*,*) *[complex,allocatable/public]*
- `fields/`**`db_lmloc`** (*,*) *[complex,pointer/public]*
- `fields/`**`db_rloc`** (*,*) *[complex,pointer/public]*
- `fields/`**`ddb_ic`** (*,*) *[complex,allocatable/public]*
- `fields/`**`ddb_ic_lmloc`** (*,*) *[complex,allocatable/public]*
- `fields/`**`ddb_lmloc`** (*,*) *[complex,pointer/public]*
- `fields/`**`ddb_rloc`** (*,*) *[complex,pointer/public]*
- `fields/`**`ddj_ic`** (*,*) *[complex,allocatable/public]*
- `fields/`**`ddj_ic_lmloc`** (*,*) *[complex,allocatable/public]*
- `fields/`**`ddj_lmloc`** (*,*) *[complex,pointer/public]*
- `fields/`**`ddj_rloc`** (*,*) *[complex,allocatable/public]*

- fields/**ddw_lmloc** (*,*) *[complex,pointer/public]*
- fields/**ddw_rloc** (*,*) *[complex,pointer/public]*
- fields/**dj_ic** (*,*) *[complex,allocatable/public]*
- fields/**dj_ic_lmloc** (*,*) *[complex,allocatable/public]*
- fields/**dj_lmloc** (*,*) *[complex,pointer/public]*
- fields/**dj_rloc** (*,*) *[complex,pointer/public]*
- fields/**dp_lmloc** (*,*) *[complex,pointer/public]*
- fields/**dp_rloc** (*,*) *[complex,pointer/public]*
- fields/**ds_lmloc** (*,*) *[complex,pointer/public]*
- fields/**ds_rloc** (*,*) *[complex,pointer/public]*
- fields/**dw_lmloc** (*,*) *[complex,pointer/public]*
- fields/**dw_rloc** (*,*) *[complex,pointer/public]*
- fields/**dxi_lmloc** (*,*) *[complex,pointer/public]*
- fields/**dxi_rloc** (*,*) *[complex,pointer/public]*
- fields/**dz_lmloc** (*,*) *[complex,pointer/public]*
- fields/**dz_rloc** (*,*) *[complex,pointer/public]*
- fields/**field_lmloc_container** (*,*,*) *[complex,target/allocatable/public]*
- fields/**field_rloc_container** (*,*,*) *[complex,target/allocatable/public]*
- fields/**flow_lmloc_container** (*,*,*) *[complex,target/allocatable/public]*
- fields/**flow_rloc_container** (*,*,*) *[complex,target/allocatable/public]*
- fields/**omega_ic** *[real,public]*
- fields/**omega_ma** *[real,public]*
- fields/**p_lmloc** (*,*) *[complex,pointer/public]*
- fields/**p_rloc** (*,*) *[complex,pointer/public]*
- fields/**phi_lmloc** (*,*) *[complex,allocatable/public]*
- fields/**phi_rloc** (*,*) *[complex,allocatable/public]*
- fields/**press_lmloc_container** (*,*,*) *[complex,target/allocatable/public]*
- fields/**press_rloc_container** (*,*,*) *[complex,target/allocatable/public]*
- fields/**s_lmloc** (*,*) *[complex,pointer/public]*
- fields/**s_lmloc_container** (*,*,*) *[complex,target/allocatable/public]*
- fields/**s_rloc** (*,*) *[complex,pointer/public]*
- fields/**s_rloc_container** (*,*,*) *[complex,target/allocatable/public]*
- fields/**w_lmloc** (*,*) *[complex,pointer/public]*
- fields/**w_rloc** (*,*) *[complex,pointer/public]*
- fields/**work_lmloc** (*,*) *[complex,allocatable/public]*
  Needed in update routines

- fields/**xi_lmloc** (*,*) *[complex,pointer/public]*
- fields/**xi_lmloc_container** (*,*,*) *[complex,target/allocatable/public]*
- fields/**xi_rloc** (*,*) *[complex,pointer/public]*
- fields/**xi_rloc_container** (*,*,*) *[complex,target/allocatable/public]*
- fields/**z_lmloc** (*,*) *[complex,pointer/public]*
- fields/**z_rloc** (*,*) *[complex,pointer/public]*

Subroutines and functions

subroutine fields/**initialize_fields**()
This subroutine allocates the different fields used in MagIC

Called from *magic*

subroutine fields/**finalize_fields**()
This subroutine deallocates the field arrays used in MagIC

Called from *magic*

10.2.7 dt_fieldsLast.f90

Description

This module contains all the work arrays of the previous time-steps needed to time advance the code. They are needed in the time-stepping scheme.

Quick access

Variables *dbdt*, *dbdt_cmb_lmloc*, *dbdt_ic*, *dbdt_lmloc_container*, *dbdt_rloc*, *dbdt_rloc_container*, *dflowdt_lmloc_container*, *dflowdt_rloc_container*, *djdt*, *djdt_ic*, *djdt_rloc*, *domega_ic_dt*, *domega_ma_dt*, *dpdt*, *dpdt_rloc*, *dphidt*, *dphidt_rloc*, *dsdt*, *dsdt_lmloc_container*, *dsdt_rloc*, *dsdt_rloc_container*, *dvsrlm_lmloc*, *dvsrlm_rloc*, *dvxbhlm_lmloc*, *dvxbhlm_rloc*, *dvxirlm_lmloc*, *dvxirlm_rloc*, *dvxvhlm_lmloc*, *dvxvhlm_rloc*, *dwdt*, *dwdt_rloc*, *dxidt*, *dxidt_lmloc_container*, *dxidt_rloc*, *dxidt_rloc_container*, *dzdt*, *dzdt_rloc*, *lorentz_torque_ic_dt*, *lorentz_torque_ma_dt*

Routines *finalize_fieldslast()*, *initialize_fieldslast()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *truncation* (*n_r_max()*, *lm_max()*, *n_r_maxmag()*, *lm_maxmag()*, *n_r_ic_maxmag()*, *fd_order()*, *fd_order_bound()*): This module defines the grid points and the truncation
- *blocking* (*llm()*, *ulm()*, *llmmag()*, *ulmmag()*): Module containing blocking information
- *logic* (*l_chemical_conv()*, *l_heat()*, *l_mag()*, *l_cond_ic()*, *l_double_curl()*, *l_rms()*, *l_finite_diff()*, *l_parallel_solve()*, *l_mag_par_solve()*, *l_phase_field()*): Module containing the logicals that control the run

- *constants* (*zero()*): module containing constants and parameters used in the code.
- *radial_data* (*nrstart()*, *nrstop()*, *nrstartmag()*, *nrstopmag()*): This module defines the MPI decomposition in the radial direction.
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *time_array*: This module defines two types that are defined to store the implicit/explicit terms at the different sub-stage/step.

Variables

- fieldslast/**dbdt** *[type_tarray,public]*
- fieldslast/**dbdt_cmb_lmloc** (*) *[complex,allocatable/public]*
- fieldslast/**dbdt_ic** *[type_tarray,public]*
- fieldslast/**dbdt_lmloc_container** (*,*,*,*) *[complex,target/allocatable/public]*
- fieldslast/**dbdt_rloc** (*,*) *[complex,pointer/public]*
- fieldslast/**dbdt_rloc_container** (*,*,*) *[complex,target/allocatable/public]*
- fieldslast/**dflowdt_lmloc_container** (*,*,*,*) *[complex,target/allocatable/public]*
- fieldslast/**dflowdt_rloc_container** (*,*,*) *[complex,target/allocatable/public]*
- fieldslast/**djdt** *[type_tarray,public]*
- fieldslast/**djdt_ic** *[type_tarray,public]*
- fieldslast/**djdt_rloc** (*,*) *[complex,pointer/public]*
- fieldslast/**domega_ic_dt** *[type_tscalar,public]*
- fieldslast/**domega_ma_dt** *[type_tscalar,public]*
- fieldslast/**dpdt** *[type_tarray,public]*
- fieldslast/**dpdt_rloc** (*,*) *[complex,pointer/public]*
- fieldslast/**dphidt** *[type_tarray,public]*
- fieldslast/**dphidt_rloc** (*,*) *[complex,allocatable/public]*
- fieldslast/**dsdt** *[type_tarray,public]*
- fieldslast/**dsdt_lmloc_container** (*,*,*,*) *[complex,target/allocatable/public]*
- fieldslast/**dsdt_rloc** (*,*) *[complex,pointer/public]*
- fieldslast/**dsdt_rloc_container** (*,*,*) *[complex,target/allocatable/public]*
- fieldslast/**dvsrlm_lmloc** (*,*,*) *[complex,pointer/public]*
- fieldslast/**dvsrlm_rloc** (*,*) *[complex,pointer/public]*
- fieldslast/**dvxbhlm_lmloc** (*,*,*) *[complex,pointer/public]*
- fieldslast/**dvxbhlm_rloc** (*,*) *[complex,pointer/public]*
- fieldslast/**dvxirlm_lmloc** (*,*,*) *[complex,pointer/public]*
- fieldslast/**dvxirlm_rloc** (*,*) *[complex,pointer/public]*
- fieldslast/**dvxvhlm_lmloc** (*,*,*) *[complex,pointer/public]*

- fieldslast/**dvxvhlm_rloc** (*,*) *[complex,pointer/public]*
- fieldslast/**dwdt** *[type_tarray,public]*
- fieldslast/**dwdt_rloc** (*,*) *[complex,pointer/public]*
- fieldslast/**dxidt** *[type_tarray,public]*
- fieldslast/**dxidt_lmloc_container** (*,*,*,*) *[complex,target/allocatable/public]*
- fieldslast/**dxidt_rloc** (*,*) *[complex,pointer/public]*
- fieldslast/**dxidt_rloc_container** (*,*,*) *[complex,target/allocatable/public]*
- fieldslast/**dzdt** *[type_tarray,public]*
- fieldslast/**dzdt_rloc** (*,*) *[complex,pointer/public]*
- fieldslast/**lorentz_torque_ic_dt** *[type_tscalar,public]*
- fieldslast/**lorentz_torque_ma_dt** *[type_tscalar,public]*

Subroutines and functions

subroutine fieldslast/**initialize_fieldslast**(*nold*, *nexp*, *nimp*)

This routine defines all the arrays needed to time advance MagIC

Parameters

- **nold** *[integer,in]* :: Number of storage of the old state
- **nexp** *[integer,in]* :: Number of explicit states
- **nimp** *[integer,in]* :: Number of implicit states

Called from *magic*

subroutine fieldslast/**finalize_fieldslast**()

Memory deallocation of d?dt arrays.

Called from *magic*

10.2.8 output_data.f90

Description

This module contains the parameters for output control

Quick access

Variables *dt_cmb*, *dt_graph*, *dt_log*, *dt_movie*, *dt_pot*, *dt_probe*, *dt_r_field*, *dt_rst*, *dt_spec*, *dt_to*, *dt_tomovie*, *l_geo*, *l_max_cmb*, *l_max_comp*, *l_max_r*, *log_file*, *lp_file*, *m_max_modes*, *n_cmb_step*, *n_cmbs*, *n_coeff_r*, *n_coeff_r_max*, *n_graph_step*, *n_graphs*, *n_log_file*, *n_log_step*, *n_logs*, *n_movie_frames*, *n_movie_step*, *n_pot_step*, *n_pots*, *n_probe_out*, *n_probe_step*, *n_r_array*, *n_r_field_step*, *n_r_fields*, *n_r_step*, *n_rst_step*, *n_rsts*, *n_spec_step*, *n_specs*, *n_stores*, *n_t_cmb*, *n_t_graph*, *n_t_log*, *n_t_movie*, *n_t_pot*, *n_t_probe*, *n_t_r_field*, *n_t_rst*, *n_t_spec*, *n_t_to*, *n_t_tomovie*, *n_time_hits*, *n_to_step*, *n_tomovie_frames*, *n_tomovie_step*, *n_tos*, *rcut*, *rdea*, *runid*,

sdens, *t_cmb*, *t_cmb_start*, *t_cmb_stop*, *t_graph*, *t_graph_start*, *t_graph_stop*, *t_log*, *t_log_start*, *t_log_stop*, *t_movie*, *t_movie_start*, *t_movie_stop*, *t_pot*, *t_pot_start*, *t_pot_stop*, *t_probe*, *t_probe_start*, *t_probe_stop*, *t_r_field*, *t_r_field_start*, *t_r_field_stop*, *t_rst*, *t_rst_start*, *t_rst_stop*, *t_spec*, *t_spec_start*, *t_spec_stop*, *t_to*, *t_to_start*, *t_to_stop*, *t_tomovie*, *t_tomovie_start*, *t_tomovie_stop*, *tag*, *zdens*

Needed modules

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

Variables

- output_data/**dt_cmb** *[real,public]*
- output_data/**dt_graph** *[real,public]*
- output_data/**dt_log** *[real,public]*
- output_data/**dt_movie** *[real,public]*
- output_data/**dt_pot** *[real,public]*
- output_data/**dt_probe** *[real,public]*
- output_data/**dt_r_field** *[real,public]*
- output_data/**dt_rst** *[real,public]*
- output_data/**dt_spec** *[real,public]*
- output_data/**dt_to** *[real,public]*
- output_data/**dt_tomovie** *[real,public]*
- output_data/**l_geo** *[integer,public]*
  max degree for geomagnetic field seen on Earth
- output_data/**l_max_cmb** *[integer,public]*
- output_data/**l_max_comp** *[integer,public]*
  Maximum spherical harmonic degree to estimate Earth-likeness
- output_data/**l_max_r** *[integer,public]*
- output_data/**log_file** *[character,public]*
- output_data/**lp_file** *[character,public]*
- output_data/**m_max_modes** *[integer,public]*
- output_data/**n_cmb_step** *[integer,public]*
- output_data/**n_cmbs** *[integer,public]*
- output_data/**n_coeff_r** (*) *[integer,allocatable/public]*
- output_data/**n_coeff_r_max** *[integer,public]*
- output_data/**n_graph_step** *[integer,public]*
- output_data/**n_graphs** *[integer,public]*
- output_data/**n_log_file** *[integer,public]*
- output_data/**n_log_step** *[integer,public]*

- output_data/**n_logs** *[integer,public]*
- output_data/**n_movie_frames** *[integer,public]*
- output_data/**n_movie_step** *[integer,public]*
- output_data/**n_pot_step** *[integer,public]*
- output_data/**n_pots** *[integer,public]*
- output_data/**n_probe_out** *[integer,public]*
- output_data/**n_probe_step** *[integer,public]*
- output_data/**n_r_array** (100) *[integer,public]*
- output_data/**n_r_field_step** *[integer,public]*
- output_data/**n_r_fields** *[integer,public]*
- output_data/**n_r_step** *[integer,public]*
- output_data/**n_rst_step** *[integer,public]*
- output_data/**n_rsts** *[integer,public]*
- output_data/**n_spec_step** *[integer,public]*
- output_data/**n_specs** *[integer,public]*
- output_data/**n_stores** *[integer,public]*
- output_data/**n_t_cmb** *[integer,public]*
- output_data/**n_t_graph** *[integer,public]*
- output_data/**n_t_log** *[integer,public]*
- output_data/**n_t_movie** *[integer,public]*
- output_data/**n_t_pot** *[integer,public]*
- output_data/**n_t_probe** *[integer,public]*
- output_data/**n_t_r_field** *[integer,public]*
- output_data/**n_t_rst** *[integer,public]*
- output_data/**n_t_spec** *[integer,public]*
- output_data/**n_t_to** *[integer,public]*
- output_data/**n_t_tomovie** *[integer,public]*
- output_data/**n_time_hits** *[integer,parameter*=5000*]*
- output_data/**n_to_step** *[integer,public]*
- output_data/**n_tomovie_frames** *[integer,public]*
- output_data/**n_tomovie_step** *[integer,public]*
- output_data/**n_tos** *[integer,public]*
- output_data/**rcut** *[real,public]*
- output_data/**rdea** *[real,public]*
- output_data/**runid** *[character,public]*

- output_data/**sdens** *[real,public]*
  Density in s when using z-integration
- output_data/**t_cmb** (5000) *[real,public]*
- output_data/**t_cmb_start** *[real,public]*
- output_data/**t_cmb_stop** *[real,public]*
- output_data/**t_graph** (5000) *[real,public]*
- output_data/**t_graph_start** *[real,public]*
- output_data/**t_graph_stop** *[real,public]*
- output_data/**t_log** (5000) *[real,public]*
- output_data/**t_log_start** *[real,public]*
- output_data/**t_log_stop** *[real,public]*
- output_data/**t_movie** (5000) *[real,public]*
- output_data/**t_movie_start** *[real,public]*
- output_data/**t_movie_stop** *[real,public]*
- output_data/**t_pot** (5000) *[real,public]*
- output_data/**t_pot_start** *[real,public]*
- output_data/**t_pot_stop** *[real,public]*
- output_data/**t_probe** (5000) *[real,public]*
- output_data/**t_probe_start** *[real,public]*
- output_data/**t_probe_stop** *[real,public]*
- output_data/**t_r_field** (5000) *[real,public]*
- output_data/**t_r_field_start** *[real,public]*
- output_data/**t_r_field_stop** *[real,public]*
- output_data/**t_rst** (5000) *[real,public]*
- output_data/**t_rst_start** *[real,public]*
- output_data/**t_rst_stop** *[real,public]*
- output_data/**t_spec** (5000) *[real,public]*
- output_data/**t_spec_start** *[real,public]*
- output_data/**t_spec_stop** *[real,public]*
- output_data/**t_to** (5000) *[real,public]*
- output_data/**t_to_start** *[real,public]*
- output_data/**t_to_stop** *[real,public]*
- output_data/**t_tomovie** (5000) *[real,public]*
- output_data/**t_tomovie_start** *[real,public]*
- output_data/**t_tomovie_stop** *[real,public]*
- output_data/**tag** *[character,public]*

- output_data/**zdens** *[real,public]*
  Density in z when using z-integration

10.2.9 constants.f90

Description

module containing constants and parameters used in the code.

Quick access

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

Needed modules

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

Variables

- constants/**c_dt_z10_ic** *[real]*
- constants/**c_dt_z10_ma** *[real]*
- constants/**c_lorentz_ic** *[real]*
- constants/**c_lorentz_ma** *[real]*
- constants/**c_moi_ic** *[real]*
  Moment of inertia of the inner core
- constants/**c_moi_ma** *[real]*
  Moment of inertia of the mantle
- constants/**c_moi_oc** *[real]*
  Moment of inertia of the outer core
- constants/**c_z10_omega_ic** *[real]*
- constants/**c_z10_omega_ma** *[real]*
- constants/**ci** *[complex,parameter*=(0.0_cp,*1.0_cp)]*
- constants/**codeversion** *[character,parameter*='6.0'*]*
- constants/**cos36** *[real,parameter*=cos(36.0_cp**pi*/180.0_cp)*]*
- constants/**cos72** *[real,parameter*=cos(72.0_cp**pi*/180.0_cp)*]*
- constants/**four** *[real,parameter*=4.0_cp*]*
  4
- constants/**half** *[real,parameter*=0.5_cp*]*
  0.5

- constants/**mass** *[real]*
  Mass of the outer core
- constants/**one** *[real,parameter*=1.0_cp*]*
  1
- constants/**osq4pi** *[real,parameter*=1.0_cp/*sq4pi]*
- constants/**pi** *[real,parameter*=4.0_cp*atan(1.0_cp)*]*
- constants/**sin36** *[real,parameter*=sin(36.0_cp**pi*/180.0_cp)*]*
- constants/**sin60** *[real,parameter*=0.5_cp*sqrt(3.0_cp)*]*
- constants/**sin72** *[real,parameter*=sin(72.0_cp**pi*/180.0_cp)*]*
- constants/**sq4pi** *[real,parameter*=sqrt(4.0_cp**pi*)*]*
  $1/\sqrt{4\pi}$
- constants/**surf_cmb** *[real]*
  Outer boundary surface
- constants/**third** *[real,parameter*=0.3333333333333333*]*
- constants/**three** *[real,parameter*=3.0_cp*]*
  1/3
- constants/**two** *[real,parameter*=2.0_cp*]*
  2
- constants/**vol_ic** *[real]*
  Volume of the inner core
- constants/**vol_oc** *[real]*
  Volume of the outer core
- constants/**y10_norm** *[real]*
- constants/**y11_norm** *[real]*
- constants/**zero** *[complex,parameter*=(0.0_cp,*0.0_cp)*]*

10.2.10 special.f90

Description

This module contains all variables for the case of an imposed IC dipole, an imposed external magnetic field and a special boundary forcing to excite inertial modes

Quick access

Variables *amp_curr*, *amp_imp*, *amp_riic*, *amp_rima*, *b0*, *bic*, *bmax_imp*, *db0*, *ddb0*, *expo_imp*, *fac_loop*, *l_curr*, *l_imp*, *le*, *lgrenoble*, *loopradratio*, *m_riic*, *m_rima*, *n_imp*, *omega_riic*, *omega_rima*, *risymmic*, *risymmma*, *rrmp*

Routines *finalize_grenoble()*, *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

- special/**amp_curr** *[real,public]*
  Amplitude of magnetic field of current loop to be scaled by Lehnert
- special/**amp_imp** *[real,public]*
  Amplitude of the time varying osc
- special/**amp_riic** *[real,public]*
- special/**amp_rima** *[real,public]*
- special/**b0** (*) *[real,allocatable/public]*
- special/**bic** *[real,public]*
- special/**bmax_imp** *[real,public]*
  Location of maximum in g_ext/g_int
- special/**db0** (*) *[real,allocatable/public]*
- special/**ddb0** (*) *[real,allocatable/public]*
- special/**expo_imp** *[real,public]*
  Exponent for decay
- special/**fac_loop** (*) *[real,allocatable/public]*
  Array of factors for computing magnetic field for loop
- special/**l_curr** *[logical,public]*
  Switch for current loop at the equator
- special/**l_imp** *[integer,public]*
  Mode of external field (dipole,quadrupole etc.)
- special/**le** *[real,public]*
  Lehnert number defined by the magnetic field at the centre
- special/**lgrenoble** *[logical,public]*
- special/**loopradratio** *[real,public]*
  Radius ratio of outer boundary/current loop
- special/**m_riic** *[integer,public]*
- special/**m_rima** *[integer,public]*
  Order of forcing
- special/**n_imp** *[integer,public]*
  Controls external field model
- special/**omega_riic** *[real,public]*
  Amplitude and frequency of forcing at the inner boundary
- special/**omega_rima** *[real,public]*
  Amplitude and frequency of forcing at the outer boundary

- special/**risymmic** *[integer,public]*
  Symmetry of forcing: 1 (0) for eq symm (antisymm)
- special/**risymmma** *[integer,public]*
- special/**rrmp** *[real,public]*
  Magnetopause radius

Subroutines and functions

subroutine special/**initialize_grenoble**()

Called from *magic*

subroutine special/**finalize_grenoble**()

Called from *magic*

10.3 MPI related modules

10.3.1 parallel.f90

Description

This module contains the blocking information

Quick access

Variables *chunksize*, *ierr*, *n_procs*, *nr_per_rank*, *nthreads*, *rank_bn*, *rank_with_l1m0*

Routines *check_mpi_error()*, *get_openmp_blocks()*, *getblocks()*, *mpiio_setup()*, *parallel()*

Needed modules

- mpi
- omp_lib

Types

- **type** parallel_mod/**unknown_type**

  Type fields

  - **% n_per_rank** *[integer]*
  - **% nstart** *[integer]*
  - **% nstop** *[integer]*

Variables

- parallel_mod/**chunksize** *[integer]*
- parallel_mod/**ierr** *[integer]*
- parallel_mod/**n_procs** *[integer]*
- parallel_mod/**nr_per_rank** *[integer]*
- parallel_mod/**nthreads** *[integer]*
- parallel_mod/**rank_bn** *[integer]*
- parallel_mod/**rank_with_l1m0** *[integer]*

Subroutines and functions

subroutine parallel_mod/**parallel**()

Called from *magic*

subroutine parallel_mod/**check_mpi_error**(*code*)

Parameters **code** *[integer,in]*

subroutine parallel_mod/**getblocks**(*bal*, *n_points*, *n_procs*)

Parameters

- **bal** (1 +) *[load,inout]*
- **n_points** *[integer,in]*
- **n_procs** *[integer,in]*

Called from *initialize_blocking()*, *initialize_geos()*, *initialize_radial_data()*, *readstartfields_mpi()*

subroutine parallel_mod/**get_openmp_blocks**(*nstart*, *nstop*)

Parameters

- **nstart** *[integer,inout]*
- **nstop** *[integer,inout]*

Called from *parallel_solve_phase()*, *parallel_solve()*, *get_nl_rms()*, *set_imex_rhs()*, *rotate_imex()*, *fft_many()*, *ifft_many()*, get_nl(), transp_lm2r_alltoallw(), transp_r2lm_alltoallw(), lo2r_redist_wait(), r2lo_redist_start(), *prepare_mat_5()*, *transform_to_lm_space()*, *get_dr_rloc()*, *get_ddr_rloc()*, *native_qst_to_spat()*, *native_sphtor_to_spat()*, *native_sph_to_spat()*, *native_sph_to_grad_spat()*, *native_spat_to_sph()*, *native_spat_to_sph_tor()*, *prepareb_fd()*, *fill_ghosts_b()*, *updateb_fd()*, *finish_exp_mag()*, *finish_exp_mag_rdist()*, *get_mag_ic_rhs_imp()*, *assemble_mag_rloc()*, *get_mag_rhs_imp()*, *get_mag_rhs_imp_ghost()*, *preparephase_fd()*, *fill_ghosts_phi()*, *updatephase_fd()*, *get_phase_rhs_imp()*, *get_phase_rhs_imp_ghost()*, *assemble_phase_rloc()*, *prepares_fd()*, *fill_ghosts_s()*, *updates_fd()*, *finish_exp_entropy()*, *finish_exp_entropy_rdist()*, *get_entropy_rhs_imp()*,

get_entropy_rhs_imp_ghost(), *assemble_entropy_rloc()*, *preparew_fd()*, *fill_ghosts_w()*, *updatew_fd()*, *finish_exp_pol()*, *finish_exp_pol_rdist()*, *get_pol_rhs_imp()*, *get_pol_rhs_imp_ghost()*, *assemble_pol()*, *assemble_pol_rloc()*, *finish_exp_smat()*, *finish_exp_smat_rdist()*, *assemble_single()*, *get_single_rhs_imp()*, *preparexi_fd()*, *fill_ghosts_xi()*, *updatexi_fd()*, *finish_exp_comp()*, *finish_exp_comp_rdist()*, *get_comp_rhs_imp()*, *get_comp_rhs_imp_ghost()*, *assemble_comp_rloc()*, *preparez_fd()*, *fill_ghosts_z()*, *updatez_fd()*, *get_tor_rhs_imp()*, *get_tor_rhs_imp_ghost()*, *assemble_tor_rloc()*

subroutine parallel_mod/**mpiio_setup**(*info*)

This routine set ups the default MPI-IO configuration. This is based on recommandations from IDRIS "Best practices for parallel IO and MPI-IO hints"

Parameters **info** *[integer,out]*

Called from open_write_header(), *write_pot_mpi()*, *open_graph_file()*, *readstartfields_mpi()*, *store_mpi()*

10.3.2 radial_data.f90

Description

This module defines the MPI decomposition in the radial direction.

Quick access

Variables *n_r_cmb*, *n_r_icb*, *nrstart*, *nrstartmag*, *nrstop*, *nrstopmag*, *radial_balance*

Routines *finalize_radial_data()*, *initialize_radial_data()*

Needed modules

- *parallel_mod* (rank(), *n_procs()*, *nr_per_rank()*, load(), *getblocks()*): This module contains the blocking information
- *logic* (*l_mag()*, *lverbose()*, *l_finite_diff()*): Module containing the logicals that control the run

Variables

- radial_data/**n_r_cmb** *[integer,public]*
- radial_data/**n_r_icb** *[integer,public]*
- radial_data/**nrstart** *[integer,public]*
- radial_data/**nrstartmag** *[integer,public]*
- radial_data/**nrstop** *[integer,public]*
- radial_data/**nrstopmag** *[integer,public]*
- radial_data/**radial_balance** (*) *[load,allocatable/public]*

Subroutines and functions

subroutine radial_data/**initialize_radial_data**(*n_r_max*)

This subroutine is used to set up the MPI decomposition in the radial direction

Parameters **n_r_max** *[integer,in]* :: Number of radial grid points

Called from *magic*

Call to *getblocks()*

subroutine radial_data/**finalize_radial_data**()

Called from *magic*

10.3.3 communications.f90

Description

This module contains the different MPI communicators used in MagIC.

Quick access

Variables *get_global_sum*, *gt_cheb*, *gt_ic*, *gt_oc*, *reduce_radial*, *send_lm_pair_to_master*, *temp_gather_lo*

Routines *allgather_from_rloc()*, *create_gather_type()*, *destroy_gather_type()*, *finalize_communications()*, *find_faster_block()*, *find_faster_comm()*, *gather_all_from_lo_to_rank0()*, *gather_from_lo_to_rank0()*, *gather_from_rloc()*, *get_global_sum_cmplx_1d()*, *get_global_sum_cmplx_2d()*, *get_global_sum_real_2d()*, *initialize_communications()*, *lm2lo_redist()*, *lo2lm_redist()*, *myallgather()*, *reduce_radial_1d()*, *reduce_radial_2d()*, *reduce_scalar()*, *scatter_from_rank0_to_lo()*, *send_lm_pair_to_master_arr()*, *send_lm_pair_to_master_scal_cmplx()*, *send_lm_pair_to_master_scal_real()*, *send_scal_lm_to_master()*

Needed modules

- mpimod
- *constants* (*zero()*): module containing constants and parameters used in the code.
- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc* (*memwrite()*, *bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *parallel_mod* (rank(), *n_procs()*, *ierr()*): This module contains the blocking information
- *truncation* (*l_max()*, *lm_max()*, *minc()*, *n_r_max()*, *n_r_ic_max()*, *l_axi()*, *fd_order()*, *fd_order_bound()*): This module defines the grid points and the truncation
- *blocking* (*st_map()*, *lo_map()*, *lm_balance()*, *llm()*, *ulm()*): Module containing blocking information

- *radial_data* (*nrstart()*, *nrstop()*, *radial_balance()*): This module defines the MPI decomposition in the radial direction.
- *logic* (*l_mag()*, *l_conv()*, *l_heat()*, *l_chemical_conv()*, *l_finite_diff()*, *l_mag_kin()*, *l_double_curl()*, *l_save_out()*, *l_packed_transp()*, *l_parallel_solve()*, *l_mag_par_solve()*, *l_phase_field()*): Module containing the logicals that control the run
- *useful* (*abortrun()*): This module contains several useful routines.
- *output_data* (*n_log_file()*, *log_file()*): This module contains the parameters for output control
- iso_fortran_env (output_unit())
- mpi_ptop_mod (type_mpiptop()): This module contains the implementation of MPI_Isend/MPI_Irecv global transpose
- mpi_alltoall_mod (type_mpiatoav(), type_mpiatoaw()): This module contains the implementation of all-to-all global communicators
- *charmanip* (*capitalize()*): This module contains several useful routines to manipule character strings
- *num_param* (*mpi_transp()*, *mpi_packing()*): Module containing numerical and control parameters
- *mpi_transp_mod* (type_mpitransp()): This is an abstract class that will be used to define MPI transposers The actual implementation is deferred to either point-to-point (MPI_Isend and MPI_IRecv) communications or all-to-all (MPI_AlltoAll)

Types

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

  Type fields

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

Variables

- communications/**get_global_sum** *[public]*
- communications/**gt_cheb** *[gather_type,public]*
- communications/**gt_ic** *[gather_type,public]*
- communications/**gt_oc** *[gather_type,public]*
- communications/**reduce_radial** *[public]*
- communications/**send_lm_pair_to_master** *[public]*
- communications/**temp_gather_lo** (*) *[complex,private/allocatable]*

Subroutines and functions

subroutine communications/**initialize_communications**()

Called from *magic*

Call to *create_gather_type()*, *capitalize()*, *find_faster_comm()*, *memwrite()*

subroutine communications/**finalize_communications**()

Called from *magic*

Call to *destroy_gather_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]* :: So far, so good: c is zero.

Return **global_sum** *[real]*

subroutine communications/**gather_all_from_lo_to_rank0**(*self*, *arr_lo*, *arr_full*)

Parameters

- **self** *[gather_type]*
- **arr_lo** (1 - *llm* + *ulm*,self%dim2) *[complex]*
- **arr_full** (*lm_max*,self%dim2) *[complex]*

Called from *dtb_gather_lo_on_rank0()*, *fields_average()*, *write_pot_mpi()*, *write_pot()*, *write_one_field()*, *write_movie_frame()*, *output()*, *store()*

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]*

Called from *finalize_communications()*

subroutine communications/**gather_from_lo_to_rank0**(*arr_lo*, *arr_full*)

Parameters

- **arr_lo** (1 - *llm* + *ulm*) *[complex]*
- **arr_full** (*lm_max*) *[complex]*

Called from *fields_average()*, *get_e_mag()*, *write_bcmb()*, *write_coeff_r()*, *output()*, *store_mpi()*

subroutine communications/**scatter_from_rank0_to_lo**(*arr_full*, *arr_lo*)

Parameters

- **arr_full** (*lm_max*) *[complex]*
- **arr_lo** (1 - *llm* + *ulm*) *[complex]*

Called from *readstartfields_old()*, *readstartfields()*, *read_map_one_field()*, *readstartfields_mpi()*, *step_time()*

subroutine communications/**send_lm_pair_to_master_arr**(*b*, *l*, *m*, *blm0*)

Parameters

- **b** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **l** *[integer,in]*
- **m** *[integer,in]*
- **blm0** (*n_r_max*) *[complex,out]*

subroutine communications/**send_lm_pair_to_master_scal_real**(*b*, *l*, *m*, *blm0*)

Parameters

- **b** (1 - *llm* + *ulm*) *[complex,in]*

- **l** *[integer,in]*
- **m** *[integer,in]*
- **blm0** *[real,out]*

subroutine communications/**send_lm_pair_to_master_scal_cmplx**(*b*, *l*, *m*, *blm0*)

Parameters

- **b** (1 - *llm* + *ulm*) *[complex,in]*
- **l** *[integer,in]*
- **m** *[integer,in]*
- **blm0** *[complex,out]*

subroutine communications/**send_scal_lm_to_master**(*blm0*, *l*, *m*)

Parameters

- **blm0** *[real,inout]*
- **l** *[integer,in]*
- **m** *[integer,in]*

subroutine communications/**lm2lo_redist**(*arr_lmloc*, *arr_lo*)

Parameters

- **arr_lmloc** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **arr_lo** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*

Call to *abortrun()*

subroutine communications/**lo2lm_redist**(*arr_lo*, *arr_lmloc*)

Parameters

- **arr_lo** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **arr_lmloc** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*

Call to *abortrun()*

subroutine communications/**gather_from_rloc**(*arr_rloc*, *arr_glob*, *irank*)

This subroutine gather a r-distributed array on rank=irank

Parameters

- **arr_rloc** (1 - *nrstart* + *nrstop*) *[real,in]*
- **arr_glob** (*n_r_max*) *[real,out]*
- **irank** *[integer,in]*

Called from *outhelicity()*, *outphase()*, *outpar()*, *outperppar()*, *get_power()*

subroutine communications/**allgather_from_rloc**(*arr_rloc*, *arr_glob*)

This subroutine gather a r-distributed array

Parameters

- **arr_rloc** (1 - *nrstart* + *nrstop*) *[real,in]*
- **arr_glob** (*n_r_max*) *[real,out]*

Called from *get_angular_moment_rloc()*

subroutine communications/**reduce_radial_2d**(*arr_dist*, *arr_glob*, *irank*)

Parameters

- **arr_dist** (,) *[real,in]*
- **arr_glob** (,) *[real,out]*
- **irank** *[integer,in]*

subroutine communications/**reduce_radial_1d**(*arr_dist*, *arr_glob*, *irank*)

Parameters

- **arr_dist** (*) *[real,in]*
- **arr_glob** (*) *[real,inout]*
- **irank** *[integer,in]*

subroutine communications/**reduce_scalar**(*scal_dist*, *scal_glob*, *irank*)

Parameters

- **scal_dist** *[real,in]*
- **scal_glob** *[real,inout]*
- **irank** *[integer,in]*

Called from *get_e_mag()*

subroutine communications/**find_faster_block**(*idx_type*)

Parameters **idx_type** *[integer,in]*

Called from *test_lmloop()*

subroutine communications/**find_faster_comm**(*idx*, *mintime*, *n_fields*)

This subroutine tests the two MPI transposition strategies and selects the fastest one.

Parameters

- **idx** *[integer,out]*
- **mintime** *[real,out]*
- **n_fields** *[integer,in]*

Called from *initialize_communications()*

subroutine communications/**myallgather**(*arr*, *dim1*, *dim2*)

Parameters

- **arr** (dim1,dim2) *[complex,inout]*
- **dim1** *[integer,in,]*
- **dim2** *[integer,in,]*

Use *blocking*, *parallel_mod*

10.3.4 mpi_transpose.f90

Description

This is an abstract class that will be used to define MPI transposers The actual implementation is deferred to either point-to-point (MPI_Isend and MPI_IRecv) communications or all-to-all (MPI_AlltoAll)

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *truncation* (*lm_max()*, *n_r_max()*): This module defines the grid points and the truncation
- *radial_data* (*nrstart()*, *nrstop()*): This module defines the MPI decomposition in the radial direction.
- *blocking* (*llm()*, *ulm()*): Module containing blocking information

Types

- **type** mpi_transp_mod/**unknown_type**

  Type fields

  - **% n_fields** *[integer]*

10.3.5 parallel_solvers.f90

Description

This module contains the routines that are used to solve linear banded problems with R-distributed arrays.

Quick access

Routines *finalize_3()*, *finalize_5()*, *initialize_3()*, *initialize_5()*, *prepare_mat_3()*, *prepare_mat_5()*, *solver_dn_3()*, *solver_dn_5()*, *solver_finish_3()*, *solver_finish_5()*, *solver_single()*, *solver_up_3()*, *solver_up_5()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod*: This module contains the blocking information
- *radial_data* (*n_r_cmb()*, *n_r_icb()*): This module defines the MPI decomposition in the radial direction.
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *constants* (*one()*): module containing constants and parameters used in the code.
- *blocking* (*lm2l()*): Module containing blocking information
- *truncation* (*lm_max()*): This module defines the grid points and the truncation

Types

- **type** parallel_solvers/**unknown_type**

  Type fields

  - **% diag** (,) *[real,allocatable]*
  - **% lmax** *[integer]*
  - **% lmin** *[integer]*
  - **% low** (,) *[real,allocatable]*
  - **% nrmax** *[integer]*
  - **% nrmin** *[integer]*
  - **% up** (,) *[real,allocatable]*

- **type** parallel_solvers/**unknown_type**

  Type fields

  - **% diag** (,) *[real,allocatable]*
  - **% lmax** *[integer]*
  - **% lmin** *[integer]*
  - **% low1** (,) *[real,allocatable]*
  - **% low2** (,) *[real,allocatable]*
  - **% nrmax** *[integer]*
  - **% nrmin** *[integer]*
  - **% up1** (,) *[real,allocatable]*
  - **% up2** (,) *[real,allocatable]*

Subroutines and functions

subroutine parallel_solvers/**initialize_3**(*this*, *nrstart*, *nrstop*, *lmin*, *lmax*)

Memory allocation of a parallel tridiagonal matrix

Parameters

- **this** *[real]*
- **nrstart** *[integer,in]*
- **nrstop** *[integer,in]*
- **lmin** *[integer,in]*
- **lmax** *[integer,in]*

subroutine parallel_solvers/**initialize_5**(*this*, *nrstart*, *nrstop*, *lmin*, *lmax*)

Memory allocation of a parallel tridiagonal matrix

Parameters

- **this** *[real]*
- **nrstart** *[integer,in]*
- **nrstop** *[integer,in]*
- **lmin** *[integer,in]*
- **lmax** *[integer,in]*

subroutine parallel_solvers/**finalize_3**(*this*)

Memory deallocation of a parallel tridiagonal solver

Parameters **this** *[real]*

subroutine parallel_solvers/**finalize_5**(*this*)

Memory deallocation of a parallel pentadiagonal solver

Parameters **this** *[real]*

subroutine parallel_solvers/**prepare_mat_3**(*this*)

LU factorisation of a tridiagonal matrix: the diagonal is overwritten

Parameters **this** *[real]*

subroutine parallel_solvers/**prepare_mat_5**(*this*)

LU factorisation of a pentadiagonal matrix

Parameters **this** *[real]*

Call to *get_openmp_blocks()*

subroutine parallel_solvers/**solver_single**(*this*, *x*, *nrstart*, *nrstop*)

This routine is used to solve a solve one single linear system that does not depend on lm. This is for instance used for z10 when l_z10mat is required.

Parameters

- **this** *[real]*
- **x** (3 - *nrstart* + *nrstop*) *[complex,inout]*

Options

- **nrstart** *[integer,in,optional/default=(-3 - nrstop + shape(x, 0)) / (-1)]*
- **nrstop** *[integer,in,optional/default=-3 + nrstart + shape(x, 0)]*

subroutine parallel_solvers/**solver_up_3**(*this*, *x*, *lmstart*, *lmstop*, *nrstart*, *nrstop*, *tag*, *array_req*, *req*, *lms_block*, *nlm_block*)

First part of the parallel tridiag solver: forward substitution

Parameters

- **this** *[real]*
- **x** (*lm_max*,3 - *nrstart* + *nrstop*) *[complex,inout]*
- **lmstart** *[integer,in]* :: Starting lm (OMP thread dependent)
- **lmstop** *[integer,in]* :: Stopping lm (OMP thread dependent)
- **tag** *[integer,in]*
- **array_req** (*) *[integer,inout]*
- **req** *[integer,inout]*
- **lms_block** *[integer,in]* :: Starting block-index of lm
- **nlm_block** *[integer,in]* :: Size of the block

Options

- **nrstart** *[integer,in,optional/default=(-3 - nrstop + shape(x, 1)) / (-1)]* :: Starting nR
- **nrstop** *[integer,in,optional/default=-3 + nrstart + shape(x, 1)]* :: Stopping nR

subroutine parallel_solvers/**solver_up_5**(*this*, *x*, *lmstart*, *lmstop*, *nrstart*, *nrstop*, *tag*, *array_req*, *req*, *lms_block*, *nlm_block*)

First part of the parallel pentadiag solver: forward substitution

Parameters

- **this** *[real]*
- **x** (*lm_max*,5 - *nrstart* + *nrstop*) *[complex,inout]*
- **lmstart** *[integer,in]* :: Starting lm (OMP thread dependent)
- **lmstop** *[integer,in]* :: Stopping lm (OMP thread dependent)
- **tag** *[integer,in]*

- **array_req** (*) *[integer,inout]*
- **req** *[integer,inout]*
- **lms_block** *[integer,in]* :: Starting block-index of lm
- **nlm_block** *[integer,in]* :: Size of the block

Options

- **nrstart** *[integer,in,optional/default=(-5 - nrstop + shape(x, 1)) / (-1)]* :: Starting nR
- **nrstop** *[integer,in,optional/default=-5 + nrstart + shape(x, 1)]* :: Stopping nR

subroutine parallel_solvers/**solver_dn_3**(*this*, *x*, *lmstart*, *lmstop*, *nrstart*, *nrstop*, *tag*, *array_req*, *req*, *lms_block*, *nlm_block*)

Second part of the parallel tridiag solver: backward substitution

Parameters

- **this** *[real]*
- **x** (*lm_max*,3 - *nrstart* + *nrstop*) *[complex,inout]*
- **lmstart** *[integer,in]* :: Starting lm (OMP thread dependent)
- **lmstop** *[integer,in]* :: Stopping lm (OMP thread dependent)
- **tag** *[integer,in]*
- **array_req** (*) *[integer,inout]*
- **req** *[integer,inout]*
- **lms_block** *[integer,in]* :: Starting block-index of lm
- **nlm_block** *[integer,in]* :: Size of the block

Options

- **nrstart** *[integer,in,optional/default=(-3 - nrstop + shape(x, 1)) / (-1)]* :: Starting nR
- **nrstop** *[integer,in,optional/default=-3 + nrstart + shape(x, 1)]* :: Stopping nR

subroutine parallel_solvers/**solver_dn_5**(*this*, *x*, *lmstart*, *lmstop*, *nrstart*, *nrstop*, *tag*, *array_req*, *req*, *lms_block*, *nlm_block*)

Second part of the parallel pentadiagonal solver: backward substitution

Parameters

- **this** *[real]*
- **x** (*lm_max*,5 - *nrstart* + *nrstop*) *[complex,inout]*
- **lmstart** *[integer,in]* :: Starting lm (OMP thread dependent)
- **lmstop** *[integer,in]* :: Stopping lm (OMP thread dependent)
- **tag** *[integer,in]*
- **array_req** (*) *[integer,inout]*
- **req** *[integer,inout]*
- **lms_block** *[integer,in]* :: Starting block-index of lm

- **nlm_block** *[integer,in]* :: Size of the block

Options

- **nrstart** *[integer,in,optional/default=(-5 - nrstop + shape(x, 1)) / (-1)]* :: Starting nR
- **nrstop** *[integer,in,optional/default=-5 + nrstart + shape(x, 1)]* :: Stopping nR

subroutine parallel_solvers/**solver_finish_3**(*this*, *x*, *lms_block*, *nlm_block*, *nrstart*, *nrstop*, *tag*, *array_req*, *req*)

Last part of the parallel tridiag solver: halo synchronisation

Parameters

- **this** *[real]*
- **x** (*lm_max*,3 - *nrstart* + *nrstop*) *[complex,inout]*
- **lms_block** *[integer,in]* :: Starting block-index of lm
- **nlm_block** *[integer,in]* :: Size of the block
- **tag** *[integer,in]*
- **array_req** (*) *[integer,inout]*
- **req** *[integer,inout]*

Options

- **nrstart** *[integer,in,optional/default=(-3 - nrstop + shape(x, 1)) / (-1)]* :: Starting index in radius
- **nrstop** *[integer,in,optional/default=-3 + nrstart + shape(x, 1)]* :: Stopping index in radius

subroutine parallel_solvers/**solver_finish_5**(*this*, *x*, *lms_block*, *nlm_block*, *nrstart*, *nrstop*, *tag*, *array_req*, *req*)

Last part of the parallel pentadiag solver: halo synchronisation

Parameters

- **this** *[real]*
- **x** (*lm_max*,5 - *nrstart* + *nrstop*) *[complex,inout]*
- **lms_block** *[integer,in]* :: Starting block-index of lm
- **nlm_block** *[integer,in]* :: Size of the block
- **tag** *[integer,in]*
- **array_req** (*) *[integer,inout]* :: MPI requests
- **req** *[integer,inout]*

Options

- **nrstart** *[integer,in,optional/default=(-5 - nrstop + shape(x, 1)) / (-1)]* :: Starting index in radius
- **nrstop** *[integer,in,optional/default=-5 + nrstart + shape(x, 1)]* :: Stopping index in radius

10.4 Code initialization

10.4.1 Namelists.f90

Description

Read and print the input namelist

Quick access

Variables *runhours*, *runminutes*, *runseconds*

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

Needed modules

- iso_fortran_env (output_unit())
- *precision_mod*: This module controls the precision used in MagIC
- *constants*: module containing constants and parameters used in the code.
- *truncation*: This module defines the grid points and the truncation
- *radial_functions*: This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *physical_parameters*: Module containing the physical parameters
- *num_param*: Module containing numerical and control parameters
- *torsional_oscillations*: This module contains information for TO calculation and output
- *init_fields*: This module is used to construct the initial solution.
- *logic*: Module containing the logicals that control the run
- *output_data*: This module contains the parameters for output control
- *parallel_mod*: This module contains the blocking information
- *special*: This module contains all variables for the case of an imposed IC dipole, an imposed external magnetic field and a special boundary forcing to excite inertial modes
- *movie_data* (*movie()*, *n_movies()*, *n_movies_max()*)
- *charmanip* (*length_to_blank()*, *capitalize()*): This module contains several useful routines to manipule character strings
- *probe_mod*
- *useful* (*abortrun()*): This module contains several useful routines.
- *dirk_schemes* (type_dirk()): This module defines the type_dirk which inherits from the abstract type_tscheme. It actually implements all the routine required to time-advance an diagonally implicit Runge-Kutta scheme. It makes use
- *multistep_schemes* (type_multistep()): This module defines the type_multistep which inherits from the abstract type_tscheme. It actually implements all the routine required to time-advance an IMEX multistep scheme such as CN/AB2, SBDF(2,3,4),

- *time_schemes* (type_tscheme()): This module defines an abstract class type_tscheme which is employed for the time advance of the code.

Variables

- namelists/**runhours** *[integer,private]*
- namelists/**runminutes** *[integer,private]*
- namelists/**runseconds** *[integer,private]*

Subroutines and functions

subroutine namelists/**readnamelists**(*tscheme*)

Purpose of this subroutine is to read the input namelists. This program also determins logical parameters that are stored in logic.f90.

Parameters **tscheme** *[real]*

Called from *magic*

Call to *defaultnamelists()*, *abortrun()*, *length_to_blank()*, *capitalize()*, *select_tscheme()*, *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()*

subroutine namelists/**select_tscheme**(*scheme_name*, *tscheme*)

This routine determines which family of time stepper should be initiated depending on the name found in the input namelist.

Parameters

- **scheme_name** *[character,inout]* :: Name of the time scheme
- **tscheme** *[real]*

Called from *readnamelists()*

Call to *capitalize()*

10.4.2 startFiels.f90

Description

This module is used to set-up the initial starting fields. They can be obtained by reading a starting checkpoint file or by setting some starting conditions.

Quick access

Variables *botcond*, *botxicond*, *deltacond*, *deltaxicond*, *topcond*, *topxicond*

Routines *getstartfields()*

Needed modules

- *truncation*: This module defines the grid points and the truncation
- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod*: This module contains the blocking information
- *radial_data* (*n_r_cmb()*, *n_r_icb()*, *nrstart()*, *nrstop()*): This module defines the MPI decomposition in the radial direction.
- *communications* (`lo2r_one()`): This module contains the different MPI communicators used in MagIC.
- *radial_functions* (`rscheme_oc()`, *r()*, *or1()*, *alpha0()*, *dltemp0()*, *dlalpha0()*, *beta()*, *orho1()*, *temp0()*, *rho0()*, *otemp1()*, *ogrun()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *physical_parameters* (*interior_model()*, *epss()*, *imps()*, *n_r_lcr()*, *ktopv()*, *kbotv()*, *lffac()*, *imagcon()*, *thexpnb()*, *vischeatfac()*, *impxi()*): Module containing the physical parameters
- *num_param* (*dtmax()*, *alpha()*): Module containing numerical and control parameters
- *special* (*lgrenoble()*): This module contains all variables for the case of an imposed IC dipole, an imposed external magnetic field and a special boundary forcing to excite inertial modes
- *output_data* (*log_file()*, *n_log_file()*): This module contains the parameters for output control
- *blocking* (*lo_map()*, *llm()*, *ulm()*, *ulmmag()*, *llmmag()*): Module containing blocking information
- *logic* (*l_conv()*, *l_mag()*, *l_cond_ic()*, *l_heat()*, *l_srma()*, *l_sric()*, *l_mag_kin()*, *l_mag_lf()*, *l_rot_ic()*, *l_z10mat()*, *l_lcr()*, *l_rot_ma()*, *l_temperature_diff()*, *l_single_matrix()*, *l_chemical_conv()*, *l_anelastic_liquid()*, *l_save_out()*, *l_parallel_solve()*, *l_mag_par_solve()*, *l_phase_field()*): Module containing the logicals that control the run
- *init_fields* (*l_start_file()*, *init_s1()*, *init_b1()*, *tops()*, *pt_cond()*, *initv()*, *inits()*, *initb()*, *initxi()*, *ps_cond()*, *start_file()*, *init_xi1()*, *topxi()*, *xi_cond()*, *omega_ic1()*, *omega_ma1()*, *initphi()*, *init_phi()*): This module is used to construct the initial solution.
- *fields*: This module contains all the fields used in MagIC in the hybrid (LM,r) space as well as their radial derivatives. It defines both the LM-distributed arrays and the R-distributed arrays....
- *fieldslast*: This module contains all the work arrays of the previous time-steps needed to time advance the code. They are needed in the time-stepping scheme....
- *timing* (`timer_type()`): This module contains functions that are used to measure the time spent.
- *constants* (*zero()*, *c_lorentz_ma()*, *c_lorentz_ic()*, *osq4pi()*, *one()*, *two()*): module containing constants and parameters used in the code.

- *useful* (*cc2real()*, *logwrite()*): This module contains several useful routines.
- *radial_der* (*get_dr()*, *exch_ghosts()*, *bulk_to_ghost()*): Radial derivatives functions
- *readcheckpoints* (*readstartfields_mpi()*): This module contains the functions that can help reading and mapping of the restart files
- *time_schemes* (`type_tscheme()`): This module defines an abstract class type_tscheme which is employed for the time advance of the code.
- *updatewps_mod* (*get_single_rhs_imp()*): This module handles the time advance of the poloidal potential w, the pressure p and the entropy s in one single matrix per degree. It contains the computation of the implicit terms and the linear
- *updatewp_mod* (*get_pol_rhs_imp()*, *get_pol_rhs_imp_ghost()*, *w_ghost()*, *fill_ghosts_w()*, *p0_ghost()*): This module handles the time advance of the poloidal potential w and the pressure p. It contains the computation of the implicit terms and the linear solves.
- *updates_mod* (*get_entropy_rhs_imp()*, *get_entropy_rhs_imp_ghost()*, *s_ghost()*, *fill_ghosts_s()*): This module handles the time advance of the entropy s. It contains the computation of the implicit terms and the linear solves....
- *updatexi_mod* (*get_comp_rhs_imp()*, *get_comp_rhs_imp_ghost()*, *xi_ghost()*, *fill_ghosts_xi()*): This module handles the time advance of the chemical composition xi. It contains the computation of the implicit terms and the linear solves....
- *updatephi_mod* (*get_phase_rhs_imp()*, *get_phase_rhs_imp_ghost()*, *phi_ghost()*, *fill_ghosts_phi()*): This module handles the time advance of the phase field phi. It contains the computation of the implicit terms and the linear solves....
- *updatez_mod* (*get_tor_rhs_imp()*, *get_tor_rhs_imp_ghost()*, *z_ghost()*, *fill_ghosts_z()*): This module handles the time advance of the toroidal potential z It contains the computation of the implicit terms and the linear solves....
- *updateb_mod* (*get_mag_rhs_imp()*, *get_mag_ic_rhs_imp()*, *b_ghost()*, *aj_ghost()*, *get_mag_rhs_imp_ghost()*, *fill_ghosts_b()*): This module handles the time advance of the magnetic field potentials b and aj as well as the inner core counterparts b_ic and aj_ic. It contains the computation of the implicit terms and the linear

Variables

- `start_fields/`**botcond** *[real,public]*
  Conducting heat flux at the inner boundary
- `start_fields/`**botxicond** *[real,public]*
  Conducting mass flux at the inner boundary
- `start_fields/`**deltacond** *[real,public]*
  Temperature or entropy difference between boundaries
- `start_fields/`**deltaxicond** *[real,public]*
  Composition difference between boundaries
- `start_fields/`**topcond** *[real,public]*
  Conducting heat flux at the outer boundary
- `start_fields/`**topxicond** *[real,public]*
  Conducting mass flux at the outer boundary

Subroutines and functions

subroutine start_fields/**getstartfields**(*time*, *tscheme*, *n_time_step*)

Purpose of this subroutine is to initialize the fields and other auxiliary parameters.

Parameters

- **time** *[real,out]* :: Time of the restart
- **tscheme** *[real]*
- **n_time_step** *[integer,out]* :: Number of past iterations

Called from *magic*

Call to *pt_cond()*, *ps_cond()*, *xi_cond()*, *readstartfields_old()*, *readstartfields_mpi()*, *logwrite()*, *initb()*, *initv()*, *inits()*, *initxi()*, *initphi()*, *bulk_to_ghost()*, *exch_ghosts()*, *fill_ghosts_xi()*, *get_comp_rhs_imp_ghost()*, *get_comp_rhs_imp()*, *fill_ghosts_phi()*, *get_phase_rhs_imp_ghost()*, *get_phase_rhs_imp()*, *get_single_rhs_imp()*, *fill_ghosts_s()*, *get_entropy_rhs_imp_ghost()*, *get_entropy_rhs_imp()*, *fill_ghosts_w()*, *get_pol_rhs_imp_ghost()*, *get_pol_rhs_imp()*, *fill_ghosts_z()*, *get_tor_rhs_imp_ghost()*, *get_tor_rhs_imp()*, *fill_ghosts_b()*, *get_mag_rhs_imp_ghost()*, *get_mag_rhs_imp()*, *get_mag_ic_rhs_imp()*, *cc2real()*

10.4.3 init_fields.f90

Description

This module is used to construct the initial solution.

Quick access

Variables *amp_b1*, *amp_s1*, *amp_s2*, *amp_v1*, *amp_xi1*, *amp_xi2*, *bots*, *botxi*, *bpeakbot*, *bpeaktop*, *inform*, *init_b1*, *init_phi*, *init_s1*, *init_s2*, *init_v1*, *init_xi1*, *init_xi2*, *l_reset_t*, *l_start_file*, *n_s_bounds*, *n_xi_bounds*, *nrotic*, *nrotma*, *omega_ic1*, *omega_ic2*, *omega_ma1*, *omega_ma2*, *omegaosz_ic1*, *omegaosz_ic2*, *omegaosz_ma1*, *omegaosz_ma2*, *phi_bot*, *phi_top*, *s_bot*, *s_top*, *scale_b*, *scale_s*, *scale_v*, *scale_xi*, *start_file*, *tipdipole*, *tomega_ic1*, *tomega_ic2*, *tomega_ma1*, *tomega_ma2*, *tops*, *topxi*, *tshift_ic1*, *tshift_ic2*, *tshift_ma1*, *tshift_ma2*, *xi_bot*, *xi_top*

Routines *finalize_init_fields()*, *initb()*, *initialize_init_fields()*, *initphi()*, *inits()*, *initv()*, *initxi()*, *j_cond()*, *ps_cond()*, *pt_cond()*, *xi_cond()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- iso_fortran_env (output_unit())
- *parallel_mod*: This module contains the blocking information
- mpi_ptop_mod (type_mpiptop()): This module contains the implementation of MPI_Isend/MPI_Irecv global transpose
- *mpi_transp_mod* (type_mpitransp()): This is an abstract class that will be used to define MPI transposers The actual implementation is deferred to either point-to-point (MPI_Isend and MPI_IRecv) communications or all-to-all (MPI_AlltoAll)
- *truncation* (*n_r_max()*, *n_r_maxmag()*, *n_r_ic_max()*, *lmp_max()*, *n_phi_max()*, *n_theta_max()*, *n_r_tot()*, *l_max()*, *m_max()*, *l_axi()*, *minc()*, *n_cheb_ic_max()*, *lm_max()*, *nlat_padded()*): This module defines the grid points and the truncation
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *blocking* (*lo_map()*, *st_map()*, *llm()*, *ulm()*, *llmmag()*, *ulmmag()*): Module containing blocking information
- *horizontal_data* (*sintheta()*, *dlh()*, *dtheta1s()*, *dtheta1a()*, *phi()*, *costheta()*, *hdif_b()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *logic* (*l_rot_ic()*, *l_rot_ma()*, *l_sric()*, *l_srma()*, *l_cond_ic()*, *l_temperature_diff()*, *l_chemical_conv()*, *l_anelastic_liquid()*, *l_non_adia()*, *l_finite_diff()*): Module containing the logicals that control the run
- *radial_functions* (*r_icb()*, *r()*, *r_cmb()*, *r_ic()*, *or1()*, *jvarcon()*, *lambda()*, *or2()*, *dllambda()*, *or3()*, *cheb_ic()*, *dcheb_ic()*, *d2cheb_ic()*, *cheb_norm_ic()*, *orho1()*, *chebt_ic()*, *temp0()*, *dltemp0()*, *kappa()*, *dlkappa()*, *beta()*, *dbeta()*, *epscprof()*, *ddltemp0()*, *ddlalpha0()*, *rgrav()*, *rho0()*, *dlalpha0()*, *alpha0()*, *otemp1()*, *ogrun()*, rscheme_oc(), *o_r_ic()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *radial_data* (*n_r_icb()*, *n_r_cmb()*, *nrstart()*, *nrstop()*): This module defines the MPI decomposition in the radial direction.
- *constants* (*pi()*, *y10_norm()*, *c_z10_omega_ic()*, *c_z10_omega_ma()*, *osq4pi()*, *zero()*, *one()*, *two()*, *three()*, *four()*, *third()*, *half()*, *sq4pi()*): module containing constants and parameters used in the code.
- *useful* (*random()*, *abortrun()*): This module contains several useful routines.
- *sht* (*scal_to_sh()*)
- *physical_parameters* (*imps()*, *n_imps_max()*, *n_imps()*, *phis()*, *thetas()*, *peaks()*, *widths()*, *radratio()*, *imagcon()*, *opm()*, *sigma_ratio()*, *o_sr()*, *kbots()*, *ktops()*, *opr()*, *epsc()*, *vischeatfac()*, *thexpnb()*, *tmelt()*, *impxi()*, *n_impxi_max()*, *n_impxi()*, *phixi()*, *thetaxi()*, *peakxi()*, *widthxi()*, *osc()*, *epscxi()*, *kbotxi()*, *ktopxi()*, *buofac()*, *ktopp()*, *oek()*, *epsphase()*): Module containing the physical parameters
- *algebra* (*prepare_mat()*, *solve_mat()*)
- *cosine_transform_odd*: This module contains the built-in type I discrete Cosine Transforms. This implementation is based on Numerical Recipes and FFTPACK. This only works for n_r_max-1 = 2**a 3**b 5**c, with a,b,c integers. . . .
- *dense_matrices*

- *real_matrices*
- *band_matrices*

Variables

- init_fields/**amp_b1** *[real,public]*
- init_fields/**amp_s1** *[real,public]*
- init_fields/**amp_s2** *[real,public]*
- init_fields/**amp_v1** *[real,public]*
- init_fields/**amp_xi1** *[real,public]*
- init_fields/**amp_xi2** *[real,public]*
- init_fields/**bots** (*,*) *[complex,allocatable/public]*
- init_fields/**botxi** (*,*) *[complex,allocatable/public]*
- init_fields/**bpeakbot** *[real,public]*
- init_fields/**bpeaktop** *[real,public]*
- init_fields/**inform** *[integer,public]*
  format of start_file
- init_fields/**init_b1** *[integer,public]*
- init_fields/**init_phi** *[integer,public]*
  An integer to specify phase field initial configuration
- init_fields/**init_s1** *[integer,public]*
- init_fields/**init_s2** *[integer,public]*
- init_fields/**init_v1** *[integer,public]*
- init_fields/**init_xi1** *[integer,public]*
- init_fields/**init_xi2** *[integer,public]*
- init_fields/**l_reset_t** *[logical,public]*
  reset time from startfile ?
- init_fields/**l_start_file** *[logical,public]*
  taking fields from startfile ?
- init_fields/**n_s_bounds** *[integer,parameter*=20*]*
- init_fields/**n_xi_bounds** *[integer,parameter*=20*]*
- init_fields/**nrotic** *[integer,public]*
- init_fields/**nrotma** *[integer,public]*
- init_fields/**omega_ic1** *[real,public]*
- init_fields/**omega_ic2** *[real,public]*
- init_fields/**omega_ma1** *[real,public]*
- init_fields/**omega_ma2** *[real,public]*
- init_fields/**omegaosz_ic1** *[real,public]*

- init_fields/**omegaosz_ic2** *[real,public]*
- init_fields/**omegaosz_ma1** *[real,public]*
- init_fields/**omegaosz_ma2** *[real,public]*
- init_fields/**phi_bot** *[real,public]*
  Phase field value at the inner boundary
- init_fields/**phi_top** *[real,public]*
  Phase field value at the outer boundary
- init_fields/**s_bot** (80) *[real,public]*
  input variables for tops,bots
- init_fields/**s_top** (80) *[real,public]*
- init_fields/**scale_b** *[real,public]*
- init_fields/**scale_s** *[real,public]*
- init_fields/**scale_v** *[real,public]*
- init_fields/**scale_xi** *[real,public]*
- init_fields/**start_file** *[character,public]*
  name of start_file
- init_fields/**tipdipole** *[real,public]*
  adding to symetric field
- init_fields/**tomega_ic1** *[real,public]*
- init_fields/**tomega_ic2** *[real,public]*
- init_fields/**tomega_ma1** *[real,public]*
- init_fields/**tomega_ma2** *[real,public]*
- init_fields/**tops** (*,*) *[complex,allocatable/public]*
- init_fields/**topxi** (*,*) *[complex,allocatable/public]*
- init_fields/**tshift_ic1** *[real,public]*
- init_fields/**tshift_ic2** *[real,public]*
- init_fields/**tshift_ma1** *[real,public]*
- init_fields/**tshift_ma2** *[real,public]*
- init_fields/**xi_bot** (80) *[real,public]*
  input variables for topxi,botxi
- init_fields/**xi_top** (80) *[real,public]*

Subroutines and functions

subroutine init_fields/**initialize_init_fields**()
Memory allocation

Called from *magic*

subroutine init_fields/**finalize_init_fields**()
Memory deallocation

Called from *magic*

subroutine init_fields/**initv**(*w*, *z*, *omega_ic*, *omega_ma*)

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** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **z** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **omega_ic** *[real,out]*
- **omega_ma** *[real,out]*

Called from *getstartfields()*

Call to *scal_to_sh()*, *random()*

subroutine init_fields/**inits**(*s*, *p*)

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** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **p** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*

Called from *getstartfields()*

Call to *pt_cond()*, *ps_cond()*, *random()*, *abortrun()*, *scal_to_sh()*, *prepare_mat()*

subroutine init_fields/**initxi**(*xi*)

Purpose of this subroutine is to initialize the chemical composition according to the input control parameters.

Input	value
init_xi1 < 100:	random noise initialized the noise spectrum decays as l^ (init_xi1-1) with peak amplitude amp_xi1 for l=1
init_xi1 >=100:	a specific harmonic mode initialized with amplitude amp_xi1. init_xi1 is interpreted as number llmm where ll: harmonic degree, mm: harmonic order.
init_xi2 >100 :	a second harmonic mode initialized with amplitude amp_xi2. init_xi2 is again interpreted as number llmm where ll: harmonic degree, mm: harmonic order.

Parameters **xi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*

Called from *getstartfields()*

Call to *xi_cond()*, *random()*, *abortrun()*, *scal_to_sh()*, *prepare_mat()*

subroutine init_fields/**initb**(*b*, *aj*, *b_ic*, *aj_ic*)

Purpose of this subroutine is to initialize the magnetic field according to the control parameters imagcon and init_b1/2. In addition CMB and ICB peak values are calculated for magneto convection.

Parameters

- **b** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,inout]*
- **aj** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,inout]*
- **b_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_max*) *[complex,inout]*
- **aj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_max*) *[complex,inout]*

Called from *getstartfields()*

Call to *j_cond()*, *abortrun()*, *random()*

subroutine init_fields/**initphi**(*s*, *phi*)

This subroutine sets the initial phase field distribution. It follows a tanh function with a width of size epsPhase

Parameters

- **s** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]* :: Entropy/Temperature
- **phi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]* :: Phase field

Called from *getstartfields()*

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]* :: aj(l=0,m=0) in the outer core
- **aj0_ic** (*) *[complex,out]* :: aj(l=0,m=0) in the inner core

Called from *initb()*

Call to *prepare_mat()*, *abortrun()*

subroutine init_fields/**xi_cond**(*xi0*)

Purpose of this subroutine is to solve the chemical composition equation for an the conductive (l=0,m=0)-mode. Output is the radial dependence of the solution in s0.

Parameters **xi0** (*) *[real,out]* :: spherically-symmetric part

Called from *initxi()*, *getstartfields()*

Call to *abortrun()*

subroutine init_fields/**pt_cond**(*t0*, *p0*)

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 t0 and p0.

Parameters

- **t0** (*) *[real,out]* :: spherically-symmetric temperature
- **p0** (*) *[real,out]* :: spherically-symmetric pressure

Called from *inits()*, *getstartfields()*

Call to *prepare_mat()*, *abortrun()*

subroutine init_fields/**ps_cond**(*s0*, *p0*)

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 and p0.

Parameters

- **s0** (*) *[real,out]* :: spherically-symmetric part
- **p0** (*) *[real,out]* :: spherically-symmetric part

Called from *inits()*, *getstartfields()*

Call to *prepare_mat()*, *abortrun()*

10.5 Pre-calculations

10.5.1 preCalc.f90

Quick access

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

Needed modules

- iso_fortran_env (output_unit())
- *constants*: module containing constants and parameters used in the code.
- *num_param*: Module containing numerical and control parameters
- *output_data*: This module contains the parameters for output control
- *precision_mod*: This module controls the precision used in MagIC
- *truncation* (*n_r_max()*, *l_max()*, *minc()*, *n_r_ic_max()*, *nalias()*, *n_cheb_ic_max()*, *m_max()*, *n_cheb_max()*, *lm_max()*, *n_phi_max()*, *n_theta_max()*): This module defines the grid points and the truncation
- *init_fields* (*bots()*, *tops()*, *s_bot()*, *s_top()*, *n_s_bounds()*, *l_reset_t()*, *topxi()*, *botxi()*, *xi_bot()*, *xi_top()*, *n_xi_bounds()*): This module is used to construct the initial solution.
- *parallel_mod* (rank()): This module contains the blocking information
- *logic* (*l_mag()*, *l_cond_ic()*, *l_non_rot()*, *l_mag_lf()*, *l_newmap()*, *l_anel()*, *l_heat()*, *l_anelastic_liquid()*, *l_cmb_field()*, *l_save_out()*, *l_to()*, *l_tomovie()*, *l_r_field()*, *l_movie()*, *l_lcr()*, *l_dt_cmb_field()*, *l_non_adia()*, *l_temperature_diff()*, *l_chemical_conv()*, *l_probe()*, *l_precession()*, *l_finite_diff()*, *l_full_sphere()*): Module containing the logicals that control the run
- *radial_functions* (rscheme_oc(), *temp0()*, *r_cmb()*, *ogrun()*, *r_surface()*, *visc()*, *or2()*, *r()*, *r_icb()*, *dltemp0()*, *beta()*, *rho0()*, *rgrav()*, *dbeta()*, *alpha0()*, *dentropy0()*, *sigma()*, *lambda()*, *dlkappa()*, *kappa()*, *dlvisc()*, *dllambda()*, *divktemp0()*, *radial()*, *transportproperties()*, *l_r()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *physical_parameters* (*nvareps()*, *pr()*, *prmag()*, *ra()*, *rascaled()*, *ek()*, *ekscaled()*, *opr()*, *opm()*, *o_sr()*, *radratio()*, *sigma_ratio()*, *corfac()*, *lffac()*, *buofac()*, *polind()*, *nvarcond()*, *nvardiff()*, *nvarvisc()*, *rho_ratio_ic()*, *rho_ratio_ma()*, *epsc()*, *epsc0()*, *ktops()*, *kbots()*, *interior_model()*, *r_lcr()*, *n_r_lcr()*, *mode()*, *tmagcon()*, *oek()*, *bn()*, *ktopxi()*, *kbotxi()*, *epscxi()*, *epscxi0()*, *sc()*, *osc()*, *chemfac()*, *raxi()*, *po()*, *prec_angle()*): Module containing the physical parameters
- *horizontal_data* (*horizontal()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *integration* (*rint_r()*): Radial integration functions
- *useful* (*logwrite()*, *abortrun()*): This module contains several useful routines.
- *special* (*l_curr()*, *fac_loop()*, *loopradratio()*, *amp_curr()*, *le()*): This module contains all variables for the case of an imposed IC dipole, an imposed external magnetic field and a special boundary forcing to excite inertial modes
- *time_schemes* (type_tscheme()): This module defines an abstract class type_tscheme which is employed for the time advance of the code.

Variables

Subroutines and functions

subroutine precalculations/**precalc**(*tscheme*)

Purpose of this subroutine is to initialize the calc values, arrays, constants that are used all over the code.

Parameters **tscheme** *[real]*

Called from *magic*

Call to *radial()*, *transportproperties()*, *horizontal()*, *rint_r()*, *abortrun()*, *logwrite()*

subroutine precalculations/**precalctimes**(*time*, *n_time_step*)

Precalc. after time, time and dt has been read from startfile.

Parameters

- **time** *[real,out]*
- **n_time_step** *[integer,out]*

Called from *magic*

Call to *get_hit_times()*

subroutine precalculations/**get_hit_times**(*t*, *n_t_max*, *n_t*, *l_t*, *t_start*, *t_stop*, *dt*, *n_tot*, *n_step*, *string_bn*, *time*, *tscale*)

This subroutine checks whether any specific times t(*) are given on input. If so, it returns their number n_r and sets l_t to true. If not, t(*) may also be defined by giving a time step dt or a number n_tot of desired output times and `t_stop>t_start`.

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]* :: Scale unit for time

Called from *precalctimes()*

Call to *abortrun()*

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]* :: Normalized OC surface :",ES14.6)') surf_cmb

Called from *magic*

Call to *abortrun()*

10.5.2 radial.f90

Description

This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

Quick access

Variables *alpha0*, *alpha1*, *alpha2*, *beta*, *cheb_ic*, *cheb_int*, *cheb_int_ic*, *cheb_norm_ic*, *chebt_ic*, *chebt_ic_even*, *d2cheb_ic*, *d2temp0*, *dbeta*, *dcheb_ic*, *ddbeta*, *ddlalpha0*, *ddltemp0*, *ddlvisc*, *dentropy0*, *divktemp0*, *dlalpha0*, *dlkappa*, *dllambda*, *dltemp0*, *dlvisc*, *dr_fac_ic*, *dr_top_ic*, *epscprof*, *jvarcon*, *kappa*, *l_r*, *lambda*, *ndd_costf1*, *ndd_costf1_ic*, *ndd_costf2_ic*, *ndi_costf1_ic*, *ndi_costf2_ic*, *o_r_ic*, *o_r_ic2*, *ogrun*, *or1*, *or2*, *or3*, *or4*, *orho1*, *orho2*, *otemp1*, *r*, *r_cmb*, *r_ic*, *r_icb*, *r_surface*, *rgrav*, *rho0*, *sigma*, *temp0*, *visc*

Routines *finalize_radial_functions()*, *getbackground()*, *getentropygradient()*, *initialize_radial_functions()*, *polynomialbackground()*, *radial()*, *transportproperties()*

Needed modules

- iso_fortran_env (output_unit())
- *truncation* (*n_r_max()*, *n_cheb_max()*, *n_r_ic_max()*, *fd_ratio()*, *fd_stretch()*, *fd_order()*, *fd_order_bound()*, *l_max()*): This module defines the grid points and the truncation
- *algebra* (*prepare_mat()*, *solve_mat()*)
- *constants* (*sq4pi()*, *one()*, *two()*, *three()*, *four()*, *half()*, *pi()*): module containing constants and parameters used in the code.
- *physical_parameters*: Module containing the physical parameters
- *logic* (*l_mag()*, *l_cond_ic()*, *l_heat()*, *l_anelastic_liquid()*, *l_isothermal()*, *l_anel()*, *l_non_adia()*, *l_centrifuge()*, *l_temperature_diff()*, *l_single_matrix()*, *l_var_l()*, *l_finite_diff()*, *l_newmap()*, *l_full_sphere()*): Module containing the logicals that control the run
- *radial_data* (*nrstart()*, *nrstop()*): This module defines the MPI decomposition in the radial direction.
- *chebyshev_polynoms_mod*

- *cosine_transform_odd*: This module contains the built-in type I discrete Cosine Transforms. This implementation is based on Numerical Recipes and FFTPACK. This only works for `n_r_max-1 = 2**a 3**b 5**c`, with a,b,c integers....
- *cosine_transform_even*
- *radial_scheme* (`type_rscheme()`): This is an abstract type that defines the radial scheme used in MagIC
- *chebyshev* (`type_cheb_odd()`)
- *finite_differences* (`type_fd()`): This module is used to calculate the radial grid when finite differences are requested
- *radial_der* (*get_dr()*): Radial derivatives functions
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *useful* (*logwrite()*, *abortrun()*): This module contains several useful routines.
- *parallel_mod* (`rank()`): This module contains the blocking information
- *output_data* (*tag()*): This module contains the parameters for output control
- *num_param* (*alph1()*, *alph2()*): Module containing numerical and control parameters

Variables

- radial_functions/**alpha0** (*) *[real,allocatable/public]*
  Thermal expansion coefficient
- radial_functions/**alpha1** *[real,public]*
  Input parameter for non-linear map to define degree of spacing (0.0:2.0)
- radial_functions/**alpha2** *[real,public]*
  Input parameter for non-linear map to define central point of different spacing (-1.0:1.0)
- radial_functions/**beta** (*) *[real,allocatable/public]*
  Inverse of density scale height drho0/rho0
- radial_functions/**cheb_ic** (*,*) *[real,allocatable/public]*
  Chebyshev polynomials for IC
- radial_functions/**cheb_int** (*) *[real,allocatable/public]*
  Array for cheb integrals
- radial_functions/**cheb_int_ic** (*) *[real,allocatable/public]*
  Array for integrals of cheb for IC
- radial_functions/**cheb_norm_ic** *[real,public]*
  Chebyshev normalisation for IC
- radial_functions/**chebt_ic** *[costf_odd_t,public]*
- radial_functions/**chebt_ic_even** *[costf_even_t,public]*
- radial_functions/**d2cheb_ic** (*,*) *[real,allocatable/public]*
  Second radial derivative cheb_ic
- radial_functions/**d2temp0** (*) *[real,allocatable/private]*
  Second rad. derivative of background temperature
- radial_functions/**dbeta** (*) *[real,allocatable/public]*
  Radial gradient of beta

- radial_functions/**dcheb_ic** (*,*) *[real,allocatable/public]*
  First radial derivative of cheb_ic
- radial_functions/**ddbeta** (*) *[real,allocatable/public]*
  2nd derivative of beta
- radial_functions/**ddlalpha0** (*) *[real,allocatable/public]*
  $d/dr(1/alpha d\alpha/dr)$
- radial_functions/**ddltemp0** (*) *[real,allocatable/public]*
  $d/dr(1/T dT/dr)$
- radial_functions/**ddlvisc** (*) *[real,allocatable/public]*
  2nd derivative of kinematic viscosity
- radial_functions/**dentropy0** (*) *[real,allocatable/public]*
  Radial gradient of background entropy
- radial_functions/**divktemp0** (*) *[real,allocatable/public]*
  Term for liquid anelastic approximation
- radial_functions/**dlalpha0** (*) *[real,allocatable/public]*
  $1/\alpha d\alpha/dr$
- radial_functions/**dlkappa** (*) *[real,allocatable/public]*
  Derivative of thermal diffusivity
- radial_functions/**dllambda** (*) *[real,allocatable/public]*
  Derivative of magnetic diffusivity
- radial_functions/**dltemp0** (*) *[real,allocatable/public]*
  Inverse of temperature scale height
- radial_functions/**dlvisc** (*) *[real,allocatable/public]*
  Derivative of kinematic viscosity
- radial_functions/**dr_fac_ic** *[real,public]*
  For IC: $2/(2r_i)$
- radial_functions/**dr_top_ic** (*) *[real,allocatable/public]*
  Derivative in real space for r=r_i
- radial_functions/**epscprof** (*) *[real,allocatable/public]*
  Sources in heat equations
- radial_functions/**jvarcon** (*) *[real,allocatable/public]*
  Analytical solution for toroidal field potential aj (see init_fields.f90)
- radial_functions/**kappa** (*) *[real,allocatable/public]*
  Thermal diffusivity
- radial_functions/**l_r** (*) *[integer,allocatable/public]*
  Variable degree with radius
- radial_functions/**lambda** (*) *[real,allocatable/public]*
  Array of magnetic diffusivity
- radial_functions/**ndd_costf1** *[integer,public]*
  Radii for transform
- radial_functions/**ndd_costf1_ic** *[integer,public]*
  Radii for transform

- radial_functions/**ndd_costf2_ic** *[integer,public]*
  Radii for transform
- radial_functions/**ndi_costf1_ic** *[integer,public]*
  Radii for transform
- radial_functions/**ndi_costf2_ic** *[integer,public]*
  Radii for transform
- radial_functions/**o_r_ic** (*) *[real,allocatable/public]*
  Inverse of IC radii
- radial_functions/**o_r_ic2** (*) *[real,allocatable/public]*
  Inverse of square of IC radii
- radial_functions/**ogrun** (*) *[real,allocatable/public]*
  $1/\Gamma$
- radial_functions/**or1** (*) *[real,allocatable/public]*
  $1/r$
- radial_functions/**or2** (*) *[real,allocatable/public]*
  $1/r^2$
- radial_functions/**or3** (*) *[real,allocatable/public]*
  $1/r^3$
- radial_functions/**or4** (*) *[real,allocatable/public]*
  $1/r^4$
- radial_functions/**orho1** (*) *[real,allocatable/public]*
  $1/\tilde{\rho}$
- radial_functions/**orho2** (*) *[real,allocatable/public]*
  $1/\tilde{\rho}^2$
- radial_functions/**otemp1** (*) *[real,allocatable/public]*
  Inverse of background temperature
- radial_functions/**r** (*) *[real,allocatable/public]*
  radii
- radial_functions/**r_cmb** *[real,public]*
  OC radius
- radial_functions/**r_ic** (*) *[real,allocatable/public]*
  IC radii
- radial_functions/**r_icb** *[real,public]*
  IC radius
- radial_functions/**r_surface** *[real,public]*
  Surface radius for extrapolation in units of (r_cmb-r_icb)
- radial_functions/**rgrav** (*) *[real,allocatable/public]*
  Buoyancy term *dtemp0/Di*
- radial_functions/**rho0** (*) *[real,allocatable/public]*
  Inverse of background density
- radial_functions/**sigma** (*) *[real,allocatable/public]*
  Electrical conductivity

- radial_functions/**temp0** (*) *[real,allocatable/public]*
  Background temperature
- radial_functions/**visc** (*) *[real,allocatable/public]*
  Kinematic viscosity

Subroutines and functions

subroutine radial_functions/**initialize_radial_functions**()
Initial memory allocation

Called from *magic*

subroutine radial_functions/**finalize_radial_functions**()
Memory deallocation of radial functions

Called from *magic*

subroutine radial_functions/**radial**()
Calculates everything needed for radial functions, transforms etc.

Called from *precalc()*

Call to *logwrite()*, *getentropygradient()*, *getbackground()*, *polynomialbackground()*, *cheb_grid()*, *get_chebs_even()*

subroutine radial_functions/**transportproperties**()
Calculates the transport properties: electrical conductivity, kinematic viscosity and thermal conductivity.

Called from *precalc()*

Call to *abortrun()*

subroutine radial_functions/**getentropygradient**()
This subroutine allows to calculate the background entropy gradient in case stable stratification is required

Called from *radial()*

subroutine radial_functions/**getbackground**(*input*, *boundaryval*, *output*[, *coeff*])

Linear solver of the form: df/dx +coeff*f= input with f(1)=boundaryVal

Parameters

- **input** (*n_r_max*) *[real,in]*
- **boundaryval** *[real,in]*
- **output** (*n_r_max*) *[real,out]*
- **coeff** (*n_r_max*) *[real,in,]*

Called from *radial()*

Call to *prepare_mat()*, *abortrun()*

subroutine radial_functions/**polynomialbackground**(*coeffdens*, *coefftemp*[, *coeffgrav*])

This subroutine allows to calculate a reference state based on an input polynomial function.

Parameters

- **coeffdens** (*) *[real,in]*

- **coefftemp** (*) *[real,in]*
- **coeffgrav** (*) *[real,in,]*

Called from *radial()*

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 *cosn2*, *cosn_theta_e2*, *costheta*, *dlh*, *dphi*, *dpl0eq*, *dtheta1a*, *dtheta1s*, *dtheta2a*, *dtheta2s*, *dtheta3a*, *dtheta3s*, *dtheta4a*, *dtheta4s*, *gauss*, *hdif_b*, *hdif_s*, *hdif_v*, *hdif_xi*, *n_theta_cal2ord*, *o_sin_theta*, *o_sin_theta_e2*, *osn1*, *osn2*, *phi*, *sintheta*, *sn2*, *theta*, *theta_ord*

Routines *finalize_horizontal_data()*, *gauleg()*, *horizontal()*, *initialize_horizontal_data()*

Needed modules

- *truncation* (*l_max()*, *lmp_max()*, *n_theta_max()*, *n_phi_max()*, *nlat_padded()*, *lm_max()*, *n_m_max()*, *minc()*, *m_max()*, *l_axi()*): 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.)
- *physical_parameters* (*ek()*): Module containing the physical parameters
- *num_param* (*difeta()*, *difnu()*, *difkap()*, *ldif()*, *ldifexp()*, *difchem()*): Module containing numerical and control parameters
- *blocking* (*lmp2l()*, *lmp2lm()*, *lm2l()*, *lm2m()*, *lo_map()*): Module containing blocking information
- *logic* (*l_non_rot()*): Module containing the logicals that control the run
- *plms_theta* (*plm_theta()*)
- *fft*: This module contains the native subroutines used to compute FFT's. They are based on the FFT99 package from Temperton: http://www.cesm.ucar.edu/models/cesm1.2/cesm/cesmBbrowser/html_code/cam/fft99.F90.html
- *constants* (*pi()*, *zero()*, *one()*, *two()*, *half()*): module containing constants and parameters used in the code.
- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC

Variables

- horizontal_data/**cosn2** (*) *[real,allocatable/public]*
- horizontal_data/**cosn_theta_e2** (*) *[real,allocatable/public]*
  $\cos\theta/\sin^2\theta$
- horizontal_data/**costheta** (*) *[real,allocatable/public]*
  $\cos\theta$
- horizontal_data/**dlh** (*) *[real,allocatable/public]*
  $\ell(\ell+1)$
- horizontal_data/**dphi** (*) *[complex,allocatable/public]*
  $\mathrm{i}m$
- horizontal_data/**dpl0eq** (*) *[real,allocatable/public]*
- horizontal_data/**dtheta1a** (*) *[real,allocatable/public]*
- horizontal_data/**dtheta1s** (*) *[real,allocatable/public]*
- horizontal_data/**dtheta2a** (*) *[real,allocatable/public]*
- horizontal_data/**dtheta2s** (*) *[real,allocatable/public]*
- horizontal_data/**dtheta3a** (*) *[real,allocatable/public]*
- horizontal_data/**dtheta3s** (*) *[real,allocatable/public]*
- horizontal_data/**dtheta4a** (*) *[real,allocatable/public]*
- horizontal_data/**dtheta4s** (*) *[real,allocatable/public]*
- horizontal_data/**gauss** (*) *[real,allocatable/public]*
- horizontal_data/**hdif_b** (*) *[real,allocatable/public]*
- horizontal_data/**hdif_s** (*) *[real,allocatable/public]*
- horizontal_data/**hdif_v** (*) *[real,allocatable/public]*
- horizontal_data/**hdif_xi** (*) *[real,allocatable/public]*
- horizontal_data/**n_theta_cal2ord** (*) *[integer,allocatable/public]*
- horizontal_data/**o_sin_theta** (*) *[real,allocatable/public]*
  $1/\sin\theta$
- horizontal_data/**o_sin_theta_e2** (*) *[real,allocatable/public]*
  $1/\sin^2\theta$
- horizontal_data/**osn1** (*) *[real,allocatable/public]*
- horizontal_data/**osn2** (*) *[real,allocatable/public]*
- horizontal_data/**phi** (*) *[real,allocatable/public]*
- horizontal_data/**sintheta** (*) *[real,allocatable/public]*
  $\sin\theta$
- horizontal_data/**sn2** (*) *[real,allocatable/public]*
- horizontal_data/**theta** (*) *[real,allocatable/public]*
  Gauss points (scrambled)
- horizontal_data/**theta_ord** (*) *[real,allocatable/public]*
  Gauss points (unscrambled)

Subroutines and functions

subroutine horizontal_data/**initialize_horizontal_data**()
Memory allocation of horizontal functions

Called from *magic*

subroutine horizontal_data/**finalize_horizontal_data**()
Memory deallocation of horizontal functions

Called from *magic*

Call to *finalize_fft()*

subroutine horizontal_data/**horizontal**()
Calculates functions of θ and ϕ, for example the Legendre functions, and functions of degree ℓ and order m of the legendres.

Called from *precalc()*

Call to *gauleg()*, *plm_theta()*, *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()*, *initialize_transforms()*

10.6 Time stepping

10.6.1 step_time.f90

Quick access

Routines *initialize_step_time()*, *start_from_another_scheme()*, *step_time()*, *transp_lmloc_to_rloc()*, *transp_rloc_to_lmloc()*, *transp_rloc_to_lmloc_io()*

Needed modules

- iso_fortran_env (output_unit())
- *fields*: This module contains all the fields used in MagIC in the hybrid (LM,r) space as well as their radial derivatives. It defines both the LM-distributed arrays and the R-distributed arrays....
- *fieldslast*: This module contains all the work arrays of the previous time-steps needed to time advance the code. They are needed in the time-stepping scheme....
- *parallel_mod*: This module contains the blocking information
- *precision_mod*: This module controls the precision used in MagIC
- *constants* (*zero()*, *one()*, *half()*): module containing constants and parameters used in the code.
- *truncation* (*n_r_max()*, *l_max()*, *l_maxmag()*, *lm_max()*, *lmp_max()*, *fd_order()*, *fd_order_bound()*): This module defines the grid points and the truncation
- *num_param* (*n_time_steps()*, *run_time_limit()*, *tend()*, *dtmax()*, *dtmin()*, *tscale()*, *dct_counter()*, *nl_counter()*, *solve_counter()*, *lm2phy_counter()*, *td_counter()*, *phy2lm_counter()*, *f_exp_counter()*): Module containing numerical and control parameters
- *radial_data* (*nrstart()*, *nrstop()*, *nrstartmag()*, *nrstopmag()*, *n_r_icb()*, *n_r_cmb()*): This module defines the MPI decomposition in the radial direction.
- *radial_der* (*get_dr_rloc()*, *get_ddr_rloc()*, *exch_ghosts()*, *bulk_to_ghost()*): Radial derivatives functions
- *radial_functions* (rscheme_oc()): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *logic* (*l_mag()*, *l_mag_lf()*, *l_dtb()*, *l_rms()*, *l_hel()*, *l_to()*, *l_tomovie()*, *l_r_field()*, *l_cmb_field()*, *l_htmovie()*, *l_dtrmagspec()*, *lverbose()*, *l_b_nl_icb()*, *l_par()*, *l_b_nl_cmb()*, *l_fluxprofs()*, *l_viscbccalc()*, *l_perppar()*, *l_ht()*, *l_dtbmovie()*, *l_heat()*, *l_conv()*, *l_movie()*, *l_runtimelimit()*, *l_save_out()*, *l_bridge_step()*, *l_dt_cmb_field()*, *l_chemical_conv()*, *l_mag_kin()*, *l_power()*, *l_double_curl()*, *l_pressgraph()*, *l_probe()*, *l_ab1()*, *l_finite_diff()*, *l_cond_ic()*, *l_single_matrix()*, *l_packed_transp()*, *l_rot_ic()*, *l_rot_ma()*, *l_cond_ma()*, *l_parallel_solve()*, *l_mag_par_solve()*, *l_phase_field()*): Module containing the logicals that control the run
- *init_fields* (*omega_ic1()*, *omega_ma1()*): This module is used to construct the initial solution.
- *radialloop* (*radialloopg()*)
- *lmloop_mod* (*lmloop()*, *finish_explicit_assembly()*, *assemble_stage()*, *finish_explicit_assembly_rdist()*, *lmloop_rdist()*, *assemble_stage_rdist()*)
- signals_mod (initialize_signals(), check_signals()): This module handles the reading/writing of the signal.TAG file which allows to communicate with MagIC during its execution
- *graphout_mod* (*open_graph_file()*, *close_graph_file()*): This module contains the subroutines that store the 3-D graphic files.
- *output_data* (*tag()*, *n_graph_step()*, *n_graphs()*, *n_t_graph()*, *t_graph()*, *n_spec_step()*, *n_specs()*, *n_t_spec()*, *t_spec()*, *n_movie_step()*, *n_movie_frames()*, *n_t_movie()*, *t_movie()*, *n_tomovie_step()*, *n_tomovie_frames()*, *n_t_tomovie()*, *t_tomovie()*, *n_pot_step()*, *n_pots()*, *n_t_pot()*, *t_pot()*, *n_rst_step()*, *n_rsts()*, *n_t_rst()*, *t_rst()*, *n_stores()*, *n_log_step()*, *n_logs()*, *n_t_log()*, *t_log()*, *n_cmb_step()*, *n_cmbs()*, *n_t_cmb()*, *t_cmb()*, *n_r_field_step()*, *n_r_fields()*, *n_t_r_field()*, *t_r_field()*, *n_to_step()*, *n_tos()*, *n_t_to()*, *t_to()*, *n_probe_step()*, *n_probe_out()*, *n_t_probe()*, *t_probe()*, *log_file()*, *n_log_file()*, *n_time_hits()*): This module contains the parameters for output control

- *updateb_mod* (*get_mag_rhs_imp()*, *get_mag_ic_rhs_imp()*, *b_ghost()*, *aj_ghost()*, *get_mag_rhs_imp_ghost()*, *fill_ghosts_b()*): This module handles the time advance of the magnetic field potentials b and aj as well as the inner core counterparts b_ic and aj_ic. It contains the computation of the implicit terms and the linear
- *updatewp_mod* (*get_pol_rhs_imp()*, *get_pol_rhs_imp_ghost()*, *w_ghost()*, *fill_ghosts_w()*, *p0_ghost()*): This module handles the time advance of the poloidal potential w and the pressure p. It contains the computation of the implicit terms and the linear solves.
- *updatewps_mod* (*get_single_rhs_imp()*): This module handles the time advance of the poloidal potential w, the pressure p and the entropy s in one single matrix per degree. It contains the computation of the implicit terms and the linear
- *updates_mod* (*get_entropy_rhs_imp()*, *get_entropy_rhs_imp_ghost()*, *s_ghost()*, *fill_ghosts_s()*): This module handles the time advance of the entropy s. It contains the computation of the implicit terms and the linear solves....
- *updatexi_mod* (*get_comp_rhs_imp()*, *get_comp_rhs_imp_ghost()*, *xi_ghost()*, *fill_ghosts_xi()*): This module handles the time advance of the chemical composition xi. It contains the computation of the implicit terms and the linear solves....
- *updatephi_mod* (*get_phase_rhs_imp()*, *get_phase_rhs_imp_ghost()*, *phi_ghost()*, *fill_ghosts_phi()*): This module handles the time advance of the phase field phi. It contains the computation of the implicit terms and the linear solves....
- *updatez_mod* (*get_tor_rhs_imp()*, *get_tor_rhs_imp_ghost()*, *z_ghost()*, *fill_ghosts_z()*): This module handles the time advance of the toroidal potential z It contains the computation of the implicit terms and the linear solves....
- *output_mod* (*output()*): This module handles the calls to the different output routines.
- *time_schemes* (`type_tscheme()`): This module defines an abstract class type_tscheme which is employed for the time advance of the code.
- *useful* (*l_correct_step()*, *logwrite()*): This module contains several useful routines.
- *communications* (`lo2r_field()`, `lo2r_flow()`, *scatter_from_rank0_to_lo()*, `lo2r_xi()`, `r2lo_flow()`, `r2lo_s()`, `r2lo_xi()`, `r2lo_field()`, `lo2r_s()`, `lo2r_press()`, `lo2r_one()`, `r2lo_one()`): This module contains the different MPI communicators used in MagIC.
- *courant_mod* (*dt_courant()*): This module handles the computation of Courant factors on grid space. It then checks whether the timestep size needs to be modified.
- *nonlinear_bcs* (*get_b_nl_bcs()*)
- *timing*: This module contains functions that are used to measure the time spent.
- *probe_mod*

Variables

Subroutines and functions

subroutine step_time_mod/**initialize_step_time**()

> **Called from** *magic*
>
> **Call to** `initialize_signals()`

subroutine step_time_mod/**step_time**(*time*, *tscheme*, *n_time_step*, *run_time_start*)

This subroutine performs the actual time-stepping.

Parameters

- **time** *[real,inout]*
- **tscheme** *[real]*
- **n_time_step** *[integer,inout]*
- **run_time_start** *[timer_type,in]*

Called from *magic*

Call to check_signals(), *logwrite()*, *l_correct_step()*, *open_graph_file()*, *transp_lmloc_to_rloc()*, *radialloopg()*, *finish_explicit_assembly_rdist()*, *transp_rloc_to_lmloc()*, *get_b_nl_bcs()*, *finish_explicit_assembly()*, *scatter_from_rank0_to_lo()*, *transp_rloc_to_lmloc_io()*, *output()*, *close_graph_file()*, *dt_courant()*, *start_from_another_scheme()*, *lmloop_rdist()*, *lmloop()*, *assemble_stage_rdist()*, *assemble_stage()*, *formattime()*

subroutine step_time_mod/**start_from_another_scheme**(*time*, *l_bridge_step*, *n_time_step*, *tscheme*)

This subroutine is used to initialize multisteps schemes whenever previous steps are not known. In that case a CN/AB2 scheme is used to bridge the missing steps.

Parameters

- **time** *[real,in]*
- **l_bridge_step** *[logical,in]*
- **n_time_step** *[integer,in]*
- **tscheme** *[real]*

Called from *step_time()*

Call to *get_single_rhs_imp()*, *bulk_to_ghost()*, *exch_ghosts()*, *fill_ghosts_w()*, *get_pol_rhs_imp_ghost()*, *get_pol_rhs_imp()*, *fill_ghosts_s()*, *get_entropy_rhs_imp_ghost()*, *get_entropy_rhs_imp()*, *fill_ghosts_z()*, *get_tor_rhs_imp_ghost()*, *get_tor_rhs_imp()*, *fill_ghosts_xi()*, *get_comp_rhs_imp_ghost()*, *get_comp_rhs_imp()*, *fill_ghosts_phi()*, *get_phase_rhs_imp_ghost()*, *get_phase_rhs_imp()*, *fill_ghosts_b()*, *get_mag_rhs_imp_ghost()*, *get_mag_rhs_imp()*, *get_mag_ic_rhs_imp()*

subroutine step_time_mod/**transp_lmloc_to_rloc**(*comm_counter*, *l_rloc*, *lpresscalc*, *lhtcalc*)

Here now comes the block where the LM distributed fields are redistributed to Rloc distribution which is needed for the radialLoop.

Parameters

- **comm_counter** *[timer_type,inout]*
- **l_rloc** *[logical,in]*
- **lpresscalc** *[logical,in]*
- **lhtcalc** *[logical,in]*

Called from *step_time()*

Call to *get_dr_rloc()*, *get_ddr_rloc()*

subroutine step_time_mod/**transp_rloc_to_lmloc**(*comm_counter*, *istage*, *lrloc*, *lpressnext*)

- MPI transposition from r-distributed to LM-distributed

Parameters

- **comm_counter** *[timer_type,inout]*
- **istage** *[integer,in]*
- **lrloc** *[logical,in]*
- **lpressnext** *[logical,in]*

Called from *step_time()*

subroutine step_time_mod/**transp_rloc_to_lmloc_io**(*lpresscalc*)

For now, most of the outputs use LM-distributed arrays as input. To handle that one has to transpose the missing fields.

Parameters **lpresscalc** *[logical,in]*

Called from *step_time()*

Call to *get_dr_rloc()*

10.6.2 courant.f90

Description

This module handles the computation of Courant factors on grid space. It then checks whether the timestep size needs to be modified.

Quick access

Variables *file_handle*

Routines *courant()*, *dt_courant()*, *finalize_courant()*, *initialize_courant()*

Needed modules

- *parallel_mod*: This module contains the blocking information
- *precision_mod*: This module controls the precision used in MagIC
- *truncation* (*n_phi_max()*, *n_theta_max()*): This module defines the grid points and the truncation
- *radial_data* (*nrstart()*, *nrstop()*): This module defines the MPI decomposition in the radial direction.
- *radial_functions* (*orho1()*, *orho2()*, *or4()*, *or2()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *physical_parameters* (*lffac()*, *opm()*): Module containing the physical parameters
- *num_param* (*delxr2()*, *delxh2()*): 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
- *logic* (*l_mag()*, *l_mag_lf()*, *l_mag_kin()*, *l_cour_alf_damp()*): Module containing the logicals that control the run
- *useful* (*logwrite()*): This module contains several useful routines.
- *constants* (*half()*, *one()*, *two()*): module containing constants and parameters used in the code.

Variables

- courant_mod/**file_handle** *[integer,private]*

Subroutines and functions

subroutine courant_mod/**initialize_courant**(*time*, *dt*, *tag*)

This subroutine opens the timestep.TAG file which stores the time step changes of MagIC.

Parameters

- **time** *[real,in]* :: time
- **dt** *[real,in]* :: time step
- **tag** *[character,in]* :: trailing of the fime

Called from *magic*

subroutine courant_mod/**finalize_courant**()

Called from *magic*

subroutine courant_mod/**courant**(*n_r*, *dtrkc*, *dthkc*, *vr*, *vt*, *vp*, *br*, *bt*, *bp*, *courfac*, *alffac*)

courant condition check: calculates Courant advection lengths in radial direction dtrkc and in horizontal direction dthkc on the local radial level n_r

for the effective velocity, the abs. sum of fluid velocity and Alfven velocity is taken

instead of the full Alfven velocity a modified Alfven velocity is employed that takes viscous and Joule damping into account. Different Courant factors are used for the fluid velocity and the such modified Alfven velocity

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** (,) *[real,in]* :: radial velocity
- **vt** (,) *[real,in]* :: longitudinal velocity
- **vp** (,) *[real,in]* :: azimuthal velocity
- **br** (,) *[real,in]* :: radial magnetic field

- **bt** (,) *[real,in]* :: longitudinal magnetic field
- **bp** (,) *[real,in]* :: azimuthal magnetic field
- **courfac** *[real,in]*
- **alffac** *[real,in]*

Called from *radialloop()*

subroutine courant_mod/**dt_courant**(*l_new_dt*, *dt*, *dt_new*, *dtmax*, *dtrkc*, *dthkc*, *time*)

This subroutine checks if the Courant criterion based on combined fluid and Alfven velocity is satisfied. It returns a new value for the time step size.

Parameters

- **l_new_dt** *[logical,out]* :: flag indicating that time step is changed (=1) or not (=0)
- **dt** *[real,in]* :: old time step size
- **dt_new** *[real,out]* :: new time step size
- **dtmax** *[real,in]* :: ”
- **dtrkc** (1 - *nrstart* + *nrstop*) *[real,in]* :: radial Courant time step as function of radial level
- **dthkc** (1 - *nrstart* + *nrstop*) *[real,in]* :: horizontal Courant time step as function of radial level
- **time** *[real,in]* :: Current time

Called from *step_time()*

Call to *logwrite()*

10.6.3 timing.f90

Description

This module contains functions that are used to measure the time spent.

Quick access

Types *unknown_type*

Routines *finalize()*, *formattime()*, *initialize()*, *start_count()*, *stop_count()*

Needed modules

- iso_fortran_env (output_unit())
- mpimod
- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod*: This module contains the blocking information

Types

- **type** timing/**unknown_type**

  Type fields

  - **% n_counts** *[integer]*
  - **% tstart** *[real]*
  - **% ttot** *[real]*

Variables

Subroutines and functions

subroutine timing/**initialize**(*this*)

Parameters **this** *[real]*

subroutine timing/**finalize**(*this*[, *message*[, *n_log_file*]])

Parameters

- **this** *[real]*
- **message** *[character,in,]*
- **n_log_file** *[integer,in,]*

Call to *formattime()*

subroutine timing/**start_count**(*this*)

Parameters **this** *[real]*

subroutine timing/**stop_count**(*this*[, *l_increment*])

Parameters

- **this** *[real]*
- **l_increment** *[logical,in,]*

subroutine timing/**formattime**(*n_out*, *message*, *time_in_sec*)

Parameters

- **n_out** *[integer,in]*
- **message** *[character,in]*
- **time_in_sec** *[real,in]*

Called from *finalize()*, *step_time()*

10.7 Time schemes

10.7.1 time_schemes.f90

Description

This module defines an abstract class type_tscheme which is employed for the time advance of the code.

Quick access

Routines *print_info()*

Needed modules

- iso_fortran_env (output_unit())
- *time_array*: This module defines two types that are defined to store the implicit/explicit terms at the different sub-stage/step.
- *logic* (*l_save_out()*): Module containing the logicals that control the run
- *output_data* (*log_file()*): This module contains the parameters for output control
- *precision_mod*: This module controls the precision used in MagIC

Types

- **type** time_schemes/**unknown_type**

  Type fields

  - **% alffac** *[real]*
  - **% courfac** *[real]*
  - **% dt** (*) *[real,allocatable]*
  - **% family** *[character]*
  - **% intfac** *[real]*
  - **% istage** *[integer]*
  - **% l_assembly** *[logical]*
  - **% l_exp_calc** (*) *[logical,allocatable]*
  - **% l_imp_calc_rhs** (*) *[logical,allocatable]*
  - **% nexp** *[integer]*
  - **% nimp** *[integer]*
  - **% nold** *[integer]*
  - **% nstages** *[integer]*
  - **% time_scheme** *[character]*
  - **% wimp_lin** (*) *[real,allocatable]*

Subroutines and functions

subroutine time_schemes/**print_info**(*this*, *n_log_file*)

This subroutine prints some informations about the time stepping scheme

Parameters

- **this** *[real]*
- **n_log_file** *[integer,in]* :: File unit of the log.TAG file

Called from *readstartfields()*, *readstartfields_mpi()*

10.7.2 multistep_schemes.f90

Description

This module defines the type_multistep which inherits from the abstract type_tscheme. It actually implements all the routine required to time-advance an IMEX multistep scheme such as CN/AB2, SBDF(2,3,4), CNLF, ...

Quick access

Routines *assemble_imex()*, *assemble_imex_scalar()*, *bridge_with_cnab2()*, *get_time_stage()*, *rotate_imex()*, *rotate_imex_scalar()*, *set_dt_array()*, *set_imex_rhs()*, *set_imex_rhs_ghost()*, *set_imex_rhs_scalar()*, *set_weights()*, *start_with_ab1()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- iso_fortran_env (output_unit())
- *parallel_mod*: This module contains the blocking information
- *num_param* (*alpha()*): Module containing numerical and control parameters
- *constants* (*one()*, *half()*, *two()*, *zero()*): module containing constants and parameters used in the code.
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *useful* (*abortrun()*): This module contains several useful routines.
- *output_data* (*log_file()*): This module contains the parameters for output control
- *logic* (*l_save_out()*): Module containing the logicals that control the run
- *time_schemes* (type_tscheme()): This module defines an abstract class type_tscheme which is employed for the time advance of the code.
- *time_array*: This module defines two types that are defined to store the implicit/explicit terms at the different sub-stage/step.

Types

- **type** multistep_schemes/**unknown_type**

  Type fields

  - **% wexp** (*) *[real,allocatable]*
  - **% wimp** (*) *[real,allocatable]*

Subroutines and functions

subroutine multistep_schemes/**initialize**(*this*, *time_scheme*, *courfac_nml*, *intfac_nml*, *alffac_nml*)

Parameters

- **this** *[real]*
- **time_scheme** *[character,inout]*
- **courfac_nml** *[real,in]*
- **intfac_nml** *[real,in]*
- **alffac_nml** *[real,in]*

subroutine multistep_schemes/**finalize**(*this*)

Parameters **this** *[real]*

subroutine multistep_schemes/**set_weights**(*this*, *lmatnext*)

Parameters

- **this** *[real]*
- **lmatnext** *[logical,inout]*

subroutine multistep_schemes/**set_dt_array**(*this*, *dt_new*, *dt_min*, *time*, *n_log_file*, *n_time_step*, *l_new_dtnext*)

This subroutine adjusts the time step

Parameters

- **this** *[real]*
- **dt_new** *[real,in]*
- **dt_min** *[real,in]*
- **time** *[real,in]*
- **n_log_file** *[integer,in]*
- **n_time_step** *[integer,in]*
- **l_new_dtnext** *[logical,in]*

Call to *abortrun()*

subroutine multistep_schemes/**set_imex_rhs**(*this*, *rhs*, *dfdt*)

This subroutine assembles the right-hand-side of an IMEX scheme

Parameters

- **this** *[real]*
- **rhs** (1 - dfdt%*llm* + dfdt%*ulm*,1 - dfdt%*nrstart* + dfdt%*nrstop*) *[complex,out]*
- **dfdt** *[type_tarray,in]*

Call to *get_openmp_blocks()*

subroutine multistep_schemes/**set_imex_rhs_ghost**(*this*, *rhs*, *dfdt*, *start_lm*, *stop_lm*, *ng*)

This subroutine assembles the right-hand-side of an IMEX scheme in case

Parameters

- **this** *[real]* :: is distributed over r
- **rhs** (1 - dfdt%*llm* + dfdt%*ulm*,1 - dfdt%*nrstart* + dfdt%*nrstop* + 2 * ng) *[complex,out]*
- **dfdt** *[type_tarray,in]*
- **start_lm** *[integer,in]* :: Starting lm index
- **stop_lm** *[integer,in]* :: Stopping lm index
- **ng** *[integer,in]* :: Number of ghosts zones

subroutine multistep_schemes/**set_imex_rhs_scalar**(*this*, *rhs*, *dfdt*)

This subroutine assembles the right-hand-side of an IMEX scheme

Parameters

- **this** *[real]*
- **rhs** *[real,out]*
- **dfdt** *[type_tscalar,in]*

subroutine multistep_schemes/**rotate_imex**(*this*, *dfdt*)

This subroutine is used to roll the time arrays from one time step

Parameters

- **this** *[real]*
- **dfdt** *[type_tarray,inout]*

Call to *get_openmp_blocks()*

subroutine multistep_schemes/**rotate_imex_scalar**(*this*, *dfdt*)

This subroutine is used to roll the time arrays from one time step

Parameters

- **this** *[real]*
- **dfdt** *[type_tscalar,inout]*

subroutine multistep_schemes/**bridge_with_cnab2**(*this*)

Parameters **this** *[real]*

subroutine multistep_schemes/**start_with_ab1**(*this*)

Parameters **this** *[real]*

subroutine multistep_schemes/**get_time_stage**(*this*, *tlast*, *tstage*)

Parameters

- **this** *[real]*
- **tlast** *[real,in]*
- **tstage** *[real,out]*

subroutine multistep_schemes/**assemble_imex**(*this*, *rhs*, *dfdt*)

Parameters

- **this** *[real]*
- **rhs** (1 - dfdt%*llm* + dfdt%*ulm*,1 - dfdt%*nrstart* + dfdt%*nrstop*) *[complex,out]*
- **dfdt** *[type_tarray,in]*

subroutine multistep_schemes/**assemble_imex_scalar**(*this*, *rhs*, *dfdt*)

Parameters

- **this** *[real]*
- **rhs** *[real,out]*
- **dfdt** *[type_tscalar,in]*

10.7.3 dirk_schemes.f90

Description

This module defines the type_dirk which inherits from the abstract type_tscheme. It actually implements all the routine required to time-advance an diagonally implicit Runge-Kutta scheme. It makes use of Butcher tables to construct the right-hand-sides.

Needed modules

- iso_fortran_env (output_unit())
- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod*: This module contains the blocking information
- *num_param* (*alpha()*): Module containing numerical and control parameters
- *constants* (*one()*, *half()*, *two()*, *zero()*): module containing constants and parameters used in the code.

- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *useful* (*abortrun()*): This module contains several useful routines.
- *logic* (*l_save_out()*): Module containing the logicals that control the run
- *output_data* (*log_file()*): This module contains the parameters for output control
- *time_schemes* (type_tscheme()): This module defines an abstract class type_tscheme which is employed for the time advance of the code.
- *time_array*: This module defines two types that are defined to store the implicit/explicit terms at the different sub-stage/step.

Types

- **type** dirk_schemes/**unknown_type**

  Type fields

  - **% butcher_ass_exp** (*) *[real,allocatable]*
  - **% butcher_ass_imp** (*) *[real,allocatable]*
  - **% butcher_c** (*) *[real,allocatable]*
  - **% butcher_exp** (,) *[real,allocatable]*
  - **% butcher_imp** (,) *[real,allocatable]*

Subroutines and functions

subroutine dirk_schemes/**initialize**(*this*, *time_scheme*, *courfac_nml*, *intfac_nml*, *alffac_nml*)

Parameters

- **this** *[real]*
- **time_scheme** *[character,inout]*
- **courfac_nml** *[real,in]*
- **intfac_nml** *[real,in]*
- **alffac_nml** *[real,in]*

subroutine dirk_schemes/**finalize**(*this*)

Parameters **this** *[real]*

subroutine dirk_schemes/**set_weights**(*this*, *lmatnext*)

Parameters

- **this** *[real]*
- **lmatnext** *[logical,inout]*

subroutine dirk_schemes/**set_dt_array**(*this*, *dt_new*, *dt_min*, *time*, *n_log_file*, *n_time_step*, *l_new_dtnext*)

This subroutine adjusts the time step

Parameters

- **this** *[real]*
- **dt_new** *[real,in]*
- **dt_min** *[real,in]*
- **time** *[real,in]*
- **n_log_file** *[integer,in]*
- **n_time_step** *[integer,in]*
- **l_new_dtnext** *[logical,in]*

Call to *abortrun()*

subroutine dirk_schemes/**set_imex_rhs**(*this*, *rhs*, *dfdt*)

This subroutine assembles the right-hand-side of an IMEX scheme

Parameters

- **this** *[real]*
- **rhs** (1 - dfdt%*llm* + dfdt%*ulm*,1 - dfdt%*nrstart* + dfdt%*nrstop*) *[complex,out]*
- **dfdt** *[type_tarray,in]*

Call to *get_openmp_blocks()*

subroutine dirk_schemes/**set_imex_rhs_ghost**(*this*, *rhs*, *dfdt*, *start_lm*, *stop_lm*, *ng*)

This subroutine assembles the right-hand-side of an IMEX scheme in case an array with ghosts zones is provided

Parameters

- **this** *[real]*
- **rhs** (1 - dfdt%*llm* + dfdt%*ulm*,1 - dfdt%*nrstart* + dfdt%*nrstop* + 2 * ng) *[complex,out]*
- **dfdt** *[type_tarray,in]*
- **start_lm** *[integer,in]*
- **stop_lm** *[integer,in]*
- **ng** *[integer,in]*

subroutine dirk_schemes/**assemble_imex**(*this*, *rhs*, *dfdt*)

This subroutine performs the assembly stage of an IMEX-RK scheme

Parameters

- **this** *[real]*
- **rhs** (1 - dfdt%*llm* + dfdt%*ulm*,1 - dfdt%*nrstart* + dfdt%*nrstop*) *[complex,out]*
- **dfdt** *[type_tarray,in]*

Call to *get_openmp_blocks()*

subroutine dirk_schemes/**set_imex_rhs_scalar**(*this*, *rhs*, *dfdt*)

This subroutine assembles the right-hand-side of an IMEX scheme

Parameters

- **this** *[real]*
- **rhs** *[real,out]*
- **dfdt** *[type_tscalar,in]*

subroutine dirk_schemes/**assemble_imex_scalar**(*this*, *rhs*, *dfdt*)

This subroutine performs the assembly stage of an IMEX-RK scheme for a scalar quantity

Parameters

- **this** *[real]*
- **rhs** *[real,out]*
- **dfdt** *[type_tscalar,in]*

subroutine dirk_schemes/**rotate_imex**(*this*, *dfdt*)

This subroutine is used to roll the time arrays from one time step

Parameters

- **this** *[real]*
- **dfdt** *[type_tarray,inout]*

subroutine dirk_schemes/**rotate_imex_scalar**(*this*, *dfdt*)

This subroutine is used to roll the time arrays from one time step

Parameters

- **this** *[real]*
- **dfdt** *[type_tscalar,inout]*

subroutine dirk_schemes/**bridge_with_cnab2**(*this*)

Parameters **this** *[real]*

subroutine dirk_schemes/**start_with_ab1**(*this*)

Parameters **this** *[real]*

subroutine dirk_schemes/**get_time_stage**(*this*, *tlast*, *tstage*)

Parameters

- **this** *[real]*

- **tlast** *[real,in]*
- **tstage** *[real,out]*

10.7.4 time_array.f90

Description

This module defines two types that are defined to store the implicit/explicit terms at the different sub-stage/step.

Quick access

Routines *finalize_scalar()*, *initialize_scalar()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *constants* (*zero()*): module containing constants and parameters used in the code.
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC

Types

- **type** time_array/**unknown_type**

  Type fields

  - **% expl** („,*) *[complex,pointer]*
  - **% impl** („,*) *[complex,allocatable]*
  - **% l_exp** *[logical]*
  - **% llm** *[integer]*
  - **% nrstart** *[integer]*
  - **% nrstop** *[integer]*
  - **% old** („,*) *[complex,allocatable]*
  - **% ulm** *[integer]*

- **type** time_array/**unknown_type**

  Type fields

  - **% expl** (*) *[real,allocatable]*
  - **% impl** (*) *[real,allocatable]*
  - **% old** (*) *[real,allocatable]*

Subroutines and functions

subroutine time_array/**initialize**(*this*, *llm*, *ulm*, *nrstart*, *nrstop*, *nold*, *nexp*, *nimp*[, *l_allocate_exp*])

Memory allocation of the arrays and initial values set to zero

Parameters

- **this** *[real]*
- **llm** *[integer,in]*
- **ulm** *[integer,in]*
- **nrstart** *[integer,in]*
- **nrstop** *[integer,in]*
- **nold** *[integer,in]*
- **nexp** *[integer,in]*
- **nimp** *[integer,in]*
- **l_allocate_exp** *[logical,in,]*

subroutine time_array/**finalize**(*this*)

Memory deallocation

Parameters **this** *[real]*

subroutine time_array/**initialize_scalar**(*this*, *nold*, *nexp*, *nimp*)

This routine initializes time stepper help arrays in case of a scalar

Parameters

- **this** *[real]*
- **nold** *[integer,in]* :: Number of old states
- **nexp** *[integer,in]* :: Number of explicit states
- **nimp** *[integer,in]* :: Number of implicit states

subroutine time_array/**finalize_scalar**(*this*)

Memory deallocation

Parameters **this** *[real]*

10.8 Linear calculation part of the time stepping (LMLoop)

10.8.1 LMLoop.f90

Quick access

Variables *array_of_requests*, *block_sze*, *n_penta*, *n_requests*, *n_tri*, *nblocks*

Routines *assemble_stage()*, *assemble_stage_rdist()*, *finalize_lmloop()*, *finish_explicit_assembly()*, *finish_explicit_assembly_rdist()*, *initialize_lmloop()*, *lmloop()*, *lmloop_rdist()*, *parallel_solve()*, *parallel_solve_phase()*, *set_block_number()*, *test_lmloop()*, *time_lm_loop()*

Needed modules

- iso_fortran_env (output_unit())
- *fields*: This module contains all the fields used in MagIC in the hybrid (LM,r) space as well as their radial derivatives. It defines both the LM-distributed arrays and the R-distributed arrays....
- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod*: This module contains the blocking information
- *constants* (*one()*): module containing constants and parameters used in the code.
- *useful* (*abortrun()*, *logwrite()*): This module contains several useful routines.
- *num_param* (*solve_counter()*): Module containing numerical and control parameters
- *mem_alloc* (*memwrite()*, *bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *truncation* (*l_max()*, *lm_max()*, *n_r_max()*, *n_r_maxmag()*, *n_r_ic_max()*, *lm_maxmag()*): This module defines the grid points and the truncation
- *radial_data* (*nrstart()*, *nrstop()*, *nrstartmag()*, *nrstopmag()*): This module defines the MPI decomposition in the radial direction.
- *blocking* (*lo_map()*, *llm()*, *ulm()*, *llmmag()*, *ulmmag()*, *st_map()*): Module containing blocking information
- *logic* (*l_mag()*, *l_conv()*, *l_heat()*, *l_single_matrix()*, *l_double_curl()*, *l_chemical_conv()*, *l_cond_ic()*, *l_update_s()*, *l_z10mat()*, *l_parallel_solve()*, *l_mag_par_solve()*, *l_phase_field()*): Module containing the logicals that control the run
- *time_array* (type_tarray(), type_tscalar()): This module defines two types that are defined to store the implicit/explicit terms at the different sub-stage/step.
- *time_schemes* (type_tscheme()): This module defines an abstract class type_tscheme which is employed for the time advance of the code.
- *timing* (timer_type()): This module contains functions that are used to measure the time spent.
- *updates_mod*: This module handles the time advance of the entropy s. It contains the computation of the implicit terms and the linear solves....
- *updatez_mod*: This module handles the time advance of the toroidal potential z It contains the computation of the implicit terms and the linear solves....

- *updatewp_mod*: This module handles the time advance of the poloidal potential w and the pressure p. It contains the computation of the implicit terms and the linear solves.
- *updatewps_mod*: This module handles the time advance of the poloidal potential w, the pressure p and the entropy s in one single matrix per degree. It contains the computation of the implicit terms and the linear
- *updateb_mod*: This module handles the time advance of the magnetic field potentials b and aj as well as the inner core counterparts b_ic and aj_ic. It contains the computation of the implicit terms and the linear
- *updatexi_mod*: This module handles the time advance of the chemical composition xi. It contains the computation of the implicit terms and the linear solves....
- *updatephi_mod*: This module handles the time advance of the phase field phi. It contains the computation of the implicit terms and the linear solves....

Variables

- lmloop_mod/**array_of_requests** (*) *[integer,private/allocatable]*
- lmloop_mod/**block_sze** *[integer,private]*
- lmloop_mod/**n_penta** *[integer,private]*
  Number of tridiagonal and pentadiagonal solvers
- lmloop_mod/**n_requests** *[integer,private]*
- lmloop_mod/**n_tri** *[integer,private]*
- lmloop_mod/**nblocks** *[integer,private]*

Subroutines and functions

subroutine lmloop_mod/**initialize_lmloop**(*tscheme*)

This subroutine handles the memory allocation of the matrices needed in the time advance of MagIC

Parameters **tscheme** *[real]* :: time scheme

Called from *magic*

Call to *initialize_updatewps()*, *initialize_updates()*, *initialize_updatewp()*, *initialize_updatexi()*, *initialize_updatephi()*, *initialize_updatez()*, *initialize_updateb()*, *memwrite()*, *set_block_number()*

subroutine lmloop_mod/**test_lmloop**(*tscheme*)

This subroutine is used to solve dummy linear problem to estimate the best blocking size. This is done once at the initialisation stage of MagIC.

Parameters **tscheme** *[real]* :: time scheme

Called from *magic*

Call to *prepares_fd()*, *preparexi_fd()*, *preparez_fd()*, *preparew_fd()*, *prepareb_fd()*, *find_faster_block()*

subroutine lmloop_mod/**finalize_lmloop**(*tscheme*)

This subroutine deallocates the matrices involved in the time advance of MagIC.

Parameters **tscheme** *[real]* :: time scheme

Called from *magic*

Call to *finalize_updatewps()*, *finalize_updates()*, *finalize_updatewp()*, *finalize_updatez()*, *finalize_updatexi()*, *finalize_updatephi()*, *finalize_updateb()*

subroutine lmloop_mod/**lmloop**(*time*, *timenext*, *tscheme*, *lmat*, *lrmsnext*, *lpressnext*, *dsdt*, *dwdt*, *dzdt*, *dpdt*, *dxidt*, *dphidt*, *dbdt*, *djdt*, *dbdt_ic*, *djdt_ic*, *domega_ma_dt*, *domega_ic_dt*, *lorentz_torque_ma_dt*, *lorentz_torque_ic_dt*, *b_nl_cmb*, *aj_nl_cmb*, *aj_nl_icb*)

This subroutine performs the actual time-stepping. It calls succesively the update routines of the various fields.

Parameters

- **time** *[real,in]*
- **timenext** *[real,in]*
- **tscheme** *[real]*
- **lmat** *[logical,in]*
- **lrmsnext** *[logical,in]*
- **lpressnext** *[logical,in]*
- **dsdt** *[type_tarray,inout]*
- **dwdt** *[type_tarray,inout]*
- **dzdt** *[type_tarray,inout]*
- **dpdt** *[type_tarray,inout]*
- **dxidt** *[type_tarray,inout]*
- **dphidt** *[type_tarray,inout]*
- **dbdt** *[type_tarray,inout]*
- **djdt** *[type_tarray,inout]*
- **dbdt_ic** *[type_tarray,inout]*
- **djdt_ic** *[type_tarray,inout]*
- **domega_ma_dt** *[type_tscalar,inout]*
- **domega_ic_dt** *[type_tscalar,inout]*
- **lorentz_torque_ma_dt** *[type_tscalar,inout]*
- **lorentz_torque_ic_dt** *[type_tscalar,inout]*
- **b_nl_cmb** (*lm_max*) *[complex,in]* :: nonlinear bc for b at CMB
- **aj_nl_cmb** (*lm_max*) *[complex,in]* :: nonlinear bc for aj at CMB
- **aj_nl_icb** (*lm_max*) *[complex,in]* :: nonlinear bc for dr aj at ICB

Called from *step_time()*

Call to *updatephi()*, *updates()*, *updatexi()*, *updatez()*, *updatewps()*, *updatewp()*, *updateb()*

subroutine lmloop_mod/**lmloop_rdist**(*time*, *timenext*, *tscheme*, *lmat*, *lrmsnext*, *lpressnext*, *dsdt*, *dwdt*, *dzdt*, *dpdt*, *dxidt*, *dphidt*, *dbdt*, *djdt*, *dbdt_ic*, *djdt_ic*, *domega_ma_dt*, *domega_ic_dt*, *lorentz_torque_ma_dt*, *lorentz_torque_ic_dt*, *b_nl_cmb*, *aj_nl_cmb*, *aj_nl_icb*)

This subroutine performs the actual time-stepping. It calls succesively the update routines of the various fields. This is used with the parallel finite difference solver.

Parameters

- **time** *[real,in]*
- **timenext** *[real,in]*
- **tscheme** *[real]*
- **lmat** *[logical,in]*
- **lrmsnext** *[logical,in]*
- **lpressnext** *[logical,in]*
- **dsdt** *[type_tarray,inout]*
- **dwdt** *[type_tarray,inout]*
- **dzdt** *[type_tarray,inout]*
- **dpdt** *[type_tarray,inout]*
- **dxidt** *[type_tarray,inout]*
- **dphidt** *[type_tarray,inout]*
- **dbdt** *[type_tarray,inout]*
- **djdt** *[type_tarray,inout]*
- **dbdt_ic** *[type_tarray,inout]*
- **djdt_ic** *[type_tarray,inout]*
- **domega_ma_dt** *[type_tscalar,inout]*
- **domega_ic_dt** *[type_tscalar,inout]*
- **lorentz_torque_ma_dt** *[type_tscalar,inout]*
- **lorentz_torque_ic_dt** *[type_tscalar,inout]*
- **b_nl_cmb** (*lm_max*) *[complex,in]* :: nonlinear bc for b at CMB
- **aj_nl_cmb** (*lm_max*) *[complex,in]* :: nonlinear bc for aj at CMB
- **aj_nl_icb** (*lm_max*) *[complex,in]* :: nonlinear bc for dr aj at ICB

Called from *step_time()*

Call to *preparephase_fd()*, *parallel_solve_phase()*, *fill_ghosts_phi()*, *updatephase_fd()*, *prepares_fd()*, *preparexi_fd()*, *preparez_fd()*, *preparew_fd()*, *prepareb_fd()*, *parallel_solve()*, *fill_ghosts_s()*, *fill_ghosts_xi()*, *fill_ghosts_z()*, *fill_ghosts_w()*, *fill_ghosts_b()*, *updates_fd()*, *updatexi_fd()*, *updatez_fd()*, *updatew_fd()*, *updateb_fd()*, *updateb()*

subroutine lmloop_mod/**finish_explicit_assembly**(*omega_ic*, *w*, *b_ic*, *aj_ic*, *dvsr_lmloc*, *dvxir_lmloc*, *dvxvh_lmloc*, *dvxbh_lmloc*, *lorentz_torque_ma*, *lorentz_torque_ic*, *dsdt*, *dxidt*, *dwdt*, *djdt*, *dbdt_ic*, *djdt_ic*, *domega_ma_dt*, *domega_ic_dt*, *lorentz_torque_ma_dt*, *lorentz_torque_ic_dt*, *tscheme*)

This subroutine is used to finish the computation of the explicit terms. This is only possible in a LM-distributed space since it mainly involves computation of radial derivatives.

Parameters

- **omega_ic** *[real,in]*
- **w** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **b_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_max*) *[complex,in]*
- **aj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_max*) *[complex,in]*
- **dvsr_lmloc** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dvxir_lmloc** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dvxvh_lmloc** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dvxbh_lmloc** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,inout]*
- **lorentz_torque_ma** *[real,in]*
- **lorentz_torque_ic** *[real,in]*
- **dsdt** *[type_tarray,inout]*
- **dxidt** *[type_tarray,inout]*
- **dwdt** *[type_tarray,inout]*
- **djdt** *[type_tarray,inout]*
- **dbdt_ic** *[type_tarray,inout]*
- **djdt_ic** *[type_tarray,inout]*
- **domega_ma_dt** *[type_tscalar,inout]*
- **domega_ic_dt** *[type_tscalar,inout]*
- **lorentz_torque_ma_dt** *[type_tscalar,inout]*
- **lorentz_torque_ic_dt** *[type_tscalar,inout]*
- **tscheme** *[real]*

Called from *step_time()*

Call to *finish_exp_comp()*, *finish_exp_smat()*, *finish_exp_entropy()*, *finish_exp_pol()*, *finish_exp_tor()*, *finish_exp_mag()*, *finish_exp_mag_ic()*

subroutine lmloop_mod/**finish_explicit_assembly_rdist**(*omega_ic*, *w*, *b_ic*, *aj_ic*, *dvsr_rloc*, *dvxir_rloc*, *dvxvh_rloc*, *dvxbh_rloc*, *lorentz_torque_ma*, *lorentz_torque_ic*, *dsdt_rloc*, *dxidt_rloc*, *dwdt_rloc*, *djdt_rloc*, *dbdt_ic*, *djdt_ic*, *domega_ma_dt*, *domega_ic_dt*, *lorentz_torque_ma_dt*, *lorentz_torque_ic_dt*, *tscheme*)

This subroutine is used to finish the computation of the explicit terms. This is the version that handles R-distributed arrays used when FD are employed.

Parameters

- **omega_ic** *[real,in]*
- **w** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,in]*
- **b_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_max*) *[complex,in]*
- **aj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_max*) *[complex,in]*
- **dvsr_rloc** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*
- **dvxir_rloc** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*
- **dvxvh_rloc** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*
- **dvxbh_rloc** (*lm_maxmag*,1 - *nrstartmag* + *nrstopmag*) *[complex,inout]*
- **lorentz_torque_ma** *[real,in]*
- **lorentz_torque_ic** *[real,in]*
- **dsdt_rloc** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*
- **dxidt_rloc** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*
- **dwdt_rloc** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*
- **djdt_rloc** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*
- **dbdt_ic** *[type_tarray,inout]*
- **djdt_ic** *[type_tarray,inout]*
- **domega_ma_dt** *[type_tscalar,inout]*
- **domega_ic_dt** *[type_tscalar,inout]*
- **lorentz_torque_ma_dt** *[type_tscalar,inout]*
- **lorentz_torque_ic_dt** *[type_tscalar,inout]*
- **tscheme** *[real]*

Called from *step_time()*

Call to *finish_exp_comp_rdist()*, *finish_exp_smat_rdist()*, *finish_exp_entropy_rdist()*, *finish_exp_pol_rdist()*, *finish_exp_tor()*, *finish_exp_mag_rdist()*, *finish_exp_mag_ic()*

subroutine lmloop_mod/**assemble_stage**(*time*, *omega_ic*, *omega_ic1*, *omega_ma*, *omega_ma1*, *dwdt*, *dzdt*, *dpdt*, *dsdt*, *dxidt*, *dphidt*, *dbdt*, *djdt*, *dbdt_ic*, *djdt_ic*, *domega_ic_dt*, *domega_ma_dt*, *lorentz_torque_ic_dt*, *lorentz_torque_ma_dt*, *lpressnext*, *lrmsnext*, *tscheme*)

This routine is used to call the different assembly stage of the different equations. This is only used for a special subset of IMEX-RK schemes that have `tscheme%l_assembly=.true.`

Parameters

- **time** *[real,in]*
- **omega_ic** *[real,inout]*

- **omega_ic1** *[real,inout]*
- **omega_ma** *[real,inout]*
- **omega_ma1** *[real,inout]*
- **dwdt** *[type_tarray,inout]*
- **dzdt** *[type_tarray,inout]*
- **dpdt** *[type_tarray,inout]*
- **dsdt** *[type_tarray,inout]*
- **dxidt** *[type_tarray,inout]*
- **dphidt** *[type_tarray,inout]*
- **dbdt** *[type_tarray,inout]*
- **djdt** *[type_tarray,inout]*
- **dbdt_ic** *[type_tarray,inout]*
- **djdt_ic** *[type_tarray,inout]*
- **domega_ic_dt** *[type_tscalar,inout]*
- **domega_ma_dt** *[type_tscalar,inout]*
- **lorentz_torque_ic_dt** *[type_tscalar,inout]*
- **lorentz_torque_ma_dt** *[type_tscalar,inout]*
- **lpressnext** *[logical,in]*
- **lrmsnext** *[logical,in]*
- **tscheme** *[real]*

Called from *step_time()*

Call to *assemble_comp()*, *assemble_phase()*, *assemble_single()*, *assemble_entropy()*, *assemble_pol()*, *assemble_tor()*, *assemble_mag()*

subroutine lmloop_mod/**assemble_stage_rdist**(*time*, *omega_ic*, *omega_ic1*, *omega_ma*, *omega_ma1*, *dwdt*, *dzdt*, *dpdt*, *dsdt*, *dxidt*, *dphidt*, *dbdt*, *djdt*, *dbdt_ic*, *djdt_ic*, *domega_ic_dt*, *domega_ma_dt*, *lorentz_torque_ic_dt*, *lorentz_torque_ma_dt*, *lpressnext*, *lrmsnext*, *tscheme*)

This routine is used to call the different assembly stage of the different equations. This is only used for a special subset of IMEX-RK schemes that have `tscheme%l_assembly=.true.`

Parameters

- **time** *[real,in]*
- **omega_ic** *[real,inout]*
- **omega_ic1** *[real,inout]*
- **omega_ma** *[real,inout]*
- **omega_ma1** *[real,inout]*
- **dwdt** *[type_tarray,inout]*
- **dzdt** *[type_tarray,inout]*

- **dpdt** *[type_tarray,inout]*
- **dsdt** *[type_tarray,inout]*
- **dxidt** *[type_tarray,inout]*
- **dphidt** *[type_tarray,inout]*
- **dbdt** *[type_tarray,inout]*
- **djdt** *[type_tarray,inout]*
- **dbdt_ic** *[type_tarray,inout]*
- **djdt_ic** *[type_tarray,inout]*
- **domega_ic_dt** *[type_tscalar,inout]*
- **domega_ma_dt** *[type_tscalar,inout]*
- **lorentz_torque_ic_dt** *[type_tscalar,inout]*
- **lorentz_torque_ma_dt** *[type_tscalar,inout]*
- **lpressnext** *[logical,in]*
- **lrmsnext** *[logical,in]*
- **tscheme** *[real]*

Called from *step_time()*

Call to *assemble_phase_rloc()*, *assemble_comp_rloc()*, *assemble_entropy_rloc()*, *assemble_pol_rloc()*, *assemble_tor_rloc()*, *assemble_mag_rloc()*, *assemble_mag()*

subroutine lmloop_mod/**parallel_solve_phase**(*block_sze*)

This subroutine handles the parallel solve of the phase field matrices. This needs to be updated before the temperature.

Parameters **block_sze** *[integer,in]* :: Size ot the LM blocks

Called from *lmloop_rdist()*

Call to *get_openmp_blocks()*, *abortrun()*

subroutine lmloop_mod/**parallel_solve**(*block_sze*)

This subroutine handles the parallel solve of the time-advance matrices. This works with R-distributed arrays (finite differences).

Parameters **block_sze** *[integer,in]* :: Size ot the LM blocks

Called from *lmloop_rdist()*, *time_lm_loop()*

Call to *get_openmp_blocks()*, *abortrun()*

function lmloop_mod/**set_block_number**(*nb*)

This routine returns the number of lm-blocks for solving the LM loop. This is adapted from xshells.

Parameters **nb** *[integer,inout]* :: No more than 2048 blocks

Return **nbo** *[integer]*

Called from *initialize_lmloop()*, *time_lm_loop()*

subroutine lmloop_mod/**find_faster_block**()

This routine is used to find the best lm-block size for MPI communication. This is adapted from xshells.

Call to *logwrite()*, *set_block_number()*, *time_lm_loop()*

function lmloop_mod/**time_lm_loop**(*nblk*, *nloops*)

This function is defined to measure the time of a call to the parallel solvers when a a number of block nblk is used. This takes the average over nloop calls.

Parameters

- **nblk** *[integer,inout]* :: Number of lm blocks
- **nloops** *[integer,in]* :: Number of calls

Return **time** *[real]*

Call to *set_block_number()*, *parallel_solve()*

10.8.2 updateWPS.f90

Description

This module handles the time advance of the poloidal potential w, the pressure p and the entropy s in one single matrix per degree. It contains the computation of the implicit terms and the linear solves.

Quick access

Variables *buo*, *cor00_fac*, *lwpsmat*, *pre*, *ps0mat*, *ps0mat_fac*, *ps0pivot*, *workb*, *workc*, *wpsmat*, *wpsmat_fac*, *wpspivot*

Routines *assemble_single()*, *finalize_updatewps()*, *finish_exp_smat()*, *finish_exp_smat_rdist()*, *get_ps0mat()*, *get_single_rhs_imp()*, *get_wpsmat()*, *initialize_updatewps()*, *updatewps()*

Needed modules

- omp_lib
- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *truncation* (*lm_max()*, *n_r_max()*, *l_max()*): This module defines the grid points and the truncation
- *radial_data* (*n_r_cmb()*, *n_r_icb()*, *nrstart()*, *nrstop()*): This module defines the MPI decomposition in the radial direction.
- *radial_functions* (*or1()*, *or2()*, *rho0()*, *rgrav()*, *r()*, *visc()*, *dlvisc()*, rscheme_oc(), *beta()*, *dbeta()*, *dlkappa()*, *dltemp0()*, *ddltemp0()*, *alpha0()*, *dlalpha0()*, *ddlalpha0()*, *ogrun()*, *kappa()*, *orho1()*, *dentropy0()*, *temp0()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *physical_parameters* (*kbotv()*, *ktopv()*, *ktops()*, *kbots()*, *ra()*, *opr()*, *vischeatfac()*, *thexpnb()*, *buofac()*, *corfac()*, *ktopp()*): Module containing the physical parameters
- *num_param* (*dct_counter()*, *solve_counter()*): Module containing numerical and control parameters
- *init_fields* (*tops()*, *bots()*): This module is used to construct the initial solution.
- *blocking* (*lo_sub_map()*, *lo_map()*, *st_sub_map()*, *llm()*, *ulm()*): Module containing blocking information
- *horizontal_data* (*hdif_v()*, *hdif_s()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *logic* (*l_update_v()*, *l_temperature_diff()*, *l_rms()*, *l_full_sphere()*): Module containing the logicals that control the run
- *rms* (*difpol2hint()*, *difpollmr()*): This module contains the calculation of the RMS force balance and induction terms.
- *rms_helpers* (*hint2pol()*): This module contains several useful subroutines required to compute RMS diagnostics
- *algebra* (*prepare_mat()*, *solve_mat()*)
- *communications* (*get_global_sum()*): This module contains the different MPI communicators used in MagIC.
- *parallel_mod* (*chunksize()*, rank(), *n_procs()*, *get_openmp_blocks()*): This module contains the blocking information
- *radial_der* (*get_dddr()*, *get_ddr()*, *get_dr()*, *get_dr_rloc()*): Radial derivatives functions
- *constants* (*zero()*, *one()*, *two()*, *three()*, *four()*, *third()*, *half()*, *pi()*, *osq4pi()*): module containing constants and parameters used in the code.
- *fields* (*work_lmloc()*): This module contains all the fields used in MagIC in the hybrid (LM,r) space as well as their radial derivatives. It defines both the LM-distributed arrays and the R-distributed arrays....
- *useful* (*abortrun()*): This module contains several useful routines.
- *time_schemes* (type_tscheme()): This module defines an abstract class type_tscheme which is employed for the time advance of the code.
- *time_array* (type_tarray()): This module defines two types that are defined to store the implicit/explicit terms at the different sub-stage/step.

Variables

- updatewps_mod/**buo** (*) *[complex,private/allocatable]*
- updatewps_mod/**cor00_fac** *[real,private]*
- updatewps_mod/**dif** (*) *[complex,private/allocatable]*
- updatewps_mod/**lwpsmat** (*) *[logical,allocatable/public]*
- updatewps_mod/**maxthreads** *[integer,private]*
- updatewps_mod/**pre** (*) *[complex,private/allocatable]*
- updatewps_mod/**ps0mat** (*,*) *[real,private/allocatable]*
- updatewps_mod/**ps0mat_fac** (*,*) *[real,private/allocatable]*
- updatewps_mod/**ps0pivot** (*) *[integer,private/allocatable]*

- updatewps_mod/**rhs1** (*,*,*) *[real,private/allocatable]*
- updatewps_mod/**workb** (*,*) *[complex,private/allocatable]*
- updatewps_mod/**workc** (*,*) *[complex,private/allocatable]*
- updatewps_mod/**wpsmat** (*,*,*) *[real,private/allocatable]*
- updatewps_mod/**wpsmat_fac** (*,*,*) *[real,private/allocatable]*
- updatewps_mod/**wpspivot** (*,*) *[integer,private/allocatable]*

Subroutines and functions

subroutine updatewps_mod/**initialize_updatewps**()

Called from *initialize_lmloop()*

subroutine updatewps_mod/**finalize_updatewps**()

Called from *finalize_lmloop()*

subroutine updatewps_mod/**updatewps**(*w*, *dw*, *ddw*, *z10*, *dwdt*, *p*, *dp*, *dpdt*, *s*, *ds*, *dsdt*, *tscheme*, *lrmsnext*)

updates the poloidal velocity potential w, the pressure p, the entropy and their radial derivatives.

Parameters

- **w** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dw** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **ddw** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **z10** (*n_r_max*) *[real,in]*
- **dwdt** *[type_tarray,inout]*
- **p** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dp** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dpdt** *[type_tarray,inout]*
- **s** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **ds** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dsdt** *[type_tarray,inout]*
- **tscheme** *[real]*
- **lrmsnext** *[logical,in]*

Called from *lmloop()*

Call to *get_ps0mat()*, *get_wpsmat()*, *get_single_rhs_imp()*

subroutine updatewps_mod/**finish_exp_smat**(*dvsrlm*, *ds_exp_last*)

Parameters

- **dvsrlm** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*

- **ds_exp_last** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*

Called from *finish_explicit_assembly()*

Call to *get_openmp_blocks()*

subroutine updatewps_mod/**finish_exp_smat_rdist**(*dvsrlm*, *ds_exp_last*)

Parameters

- **dvsrlm** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*
- **ds_exp_last** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*

Called from *finish_explicit_assembly_rdist()*

Call to *get_dr_rloc()*, *get_openmp_blocks()*

subroutine updatewps_mod/**assemble_single**(*s*, *ds*, *w*, *dw*, *ddw*, *dsdt*, *dwdt*, *dpdt*, *tscheme*, *lrmsnext*)

This routine is used to assemble the solution in case IMEX RK with an assembly stage are used

Parameters

- **s** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **ds** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **w** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dw** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **ddw** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dsdt** *[type_tarray,inout]*
- **dwdt** *[type_tarray,inout]*
- **dpdt** *[type_tarray,inout]*
- **tscheme** *[real]*
- **lrmsnext** *[logical,in]*

Called from *assemble_stage()*

Call to *abortrun()*, *get_openmp_blocks()*, *get_ddr()*, *hint2pol()*

subroutine updatewps_mod/**get_single_rhs_imp**(*s*, *ds*, *w*, *dw*, *ddw*, *p*, *dp*, *dsdt*, *dwdt*, *dpdt*, *tscheme*, *istage*, *l_calc_lin*, *lrmsnext*[, *l_in_cheb_space*])

– Input variables

Parameters

- **s** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **ds** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **w** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dw** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **ddw** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **p** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dp** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*

- **dsdt** *[type_tarray,inout]*
- **dwdt** *[type_tarray,inout]*
- **dpdt** *[type_tarray,inout]*
- **tscheme** *[real]*
- **istage** *[integer,in]*
- **l_calc_lin** *[logical,in]*
- **lrmsnext** *[logical,in]*
- **l_in_cheb_space** *[logical,in,]*

Called from *readstartfields_old()*, *readstartfields()*, *readstartfields_mpi()*, *getstartfields()*, *start_from_another_scheme()*, *updatewps()*

Call to *get_openmp_blocks()*, *get_ddr()*, *get_dddr()*, *hint2pol()*

subroutine updatewps_mod/**get_wpsmat**(*tscheme*, *l*, *hdif_vel*, *hdif_s*, *wpsmat*, *wpspivot*, *wpsmat_fac*)

Purpose of this subroutine is to contruct the time step matrix wpmat for the NS equation.

Parameters

- **tscheme** *[real]*
- **l** *[integer,in]*
- **hdif_vel** *[real,in]*
- **hdif_s** *[real,in]*
- **wpsmat** (3 * *n_r_max*,3 * *n_r_max*) *[real,out]* :: ‘)
- **wpspivot** (3 * *n_r_max*) *[integer,out]*
- **wpsmat_fac** (3 * *n_r_max*,2) *[real,out]*

Called from *updatewps()*

Call to *prepare_mat()*, *abortrun()*

subroutine updatewps_mod/**get_ps0mat**(*tscheme*, *psmat*, *pspivot*, *psmat_fac*)

Parameters

- **tscheme** *[real]*
- **psmat** (2 * *n_r_max*,2 * *n_r_max*) *[real,out]* :: entropy diffusion
- **pspivot** (2 * *n_r_max*) *[integer,out]*
- **psmat_fac** (2 * *n_r_max*,2) *[real,out]*

Called from *updatewps()*

Call to *prepare_mat()*, *abortrun()*

10.8.3 updateWP.f90

Description

This module handles the time advance of the poloidal potential w and the pressure p. It contains the computation of the implicit terms and the linear solves.

Quick access

Variables *ddddw*, *dwold*, *ellmat_fd*, *l_ellmat*, *lwpmat*, *p0_ghost*, *p0mat_fd*, *rhs0*, *size_rhs1*, *w_ghost*, *wmat_fd*, *work*, *wpmat_fac*

Routines *assemble_pol()*, *assemble_pol_rloc()*, *fill_ghosts_w()*, *finalize_updatewp()*, *finish_exp_pol()*, *finish_exp_pol_rdist()*, *get_elliptic_mat()*, *get_elliptic_mat_rdist()*, *get_p0mat()*, *get_p0mat_rdist()*, *get_pol()*, *get_pol_rhs_imp()*, *get_pol_rhs_imp_ghost()*, *get_wmat()*, *get_wmat_rdist()*, *get_wpmat()*, *initialize_updatewp()*, *preparew_fd()*, *updatew_fd()*, *updatewp()*

Needed modules

- omp_lib
- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *truncation* (*lm_max()*, *n_r_max()*, *l_max()*): This module defines the grid points and the truncation
- *radial_data* (*n_r_cmb()*, *n_r_icb()*, *nrstart()*, *nrstop()*): This module defines the MPI decomposition in the radial direction.
- *radial_functions* (*or1()*, *or2()*, *rho0()*, *rgrav()*, *visc()*, *dlvisc()*, *r()*, *alpha0()*, *temp0()*, *beta()*, *dbeta()*, *ogrun()*, rscheme_oc(), *ddlvisc()*, *ddbeta()*, *orho1()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *physical_parameters* (*kbotv()*, *ktopv()*, *ra()*, *buofac()*, *chemfac()*, *vischeatfac()*, *thexpnb()*, *ktopp()*): Module containing the physical parameters
- *num_param* (*dct_counter()*, *solve_counter()*): Module containing numerical and control parameters
- *blocking* (*lo_sub_map()*, *lo_map()*, *st_sub_map()*, *llm()*, *ulm()*, *st_map()*): Module containing blocking information
- *horizontal_data* (*hdif_v()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *logic* (*l_update_v()*, *l_chemical_conv()*, *l_rms()*, *l_double_curl()*, *l_fluxprofs()*, *l_finite_diff()*, *l_full_sphere()*, *l_heat()*, *l_parallel_solve()*): Module containing the logicals that control the run
- *rms* (*difpol2hint()*, *difpollmr()*): This module contains the calculation of the RMS force balance and induction terms.
- *communications* (*get_global_sum()*): This module contains the different MPI communicators used in MagIC.
- *parallel_mod*: This module contains the blocking information

- *rms_helpers* (*hint2pol()*): This module contains several useful subroutines required to compute RMS diagnostics
- *radial_der* (*get_dddr()*, *get_ddr()*, *get_dr()*, *get_dr_rloc()*, *get_ddddr_ghost()*, *bulk_to_ghost()*, *exch_ghosts()*): Radial derivatives functions
- *integration* (*rint_r()*): Radial integration functions
- *fields* (*work_lmloc()*, *s_rloc()*, *xi_rloc()*): This module contains all the fields used in MagIC in the hybrid (LM,r) space as well as their radial derivatives. It defines both the LM-distributed arrays and the R-distributed arrays....
- *constants* (*zero()*, *one()*, *two()*, *three()*, *four()*, *third()*, *half()*): module containing constants and parameters used in the code.
- *useful* (*abortrun()*): This module contains several useful routines.
- *time_schemes* (`type_tscheme()`): This module defines an abstract class type_tscheme which is employed for the time advance of the code.
- *time_array* (`type_tarray()`): This module defines two types that are defined to store the implicit/explicit terms at the different sub-stage/step.
- *parallel_solvers*: This module contains the routines that are used to solve linear banded problems with R-distributed arrays.
- *dense_matrices*
- *real_matrices*
- *band_matrices*

Variables

- updatewp_mod/**buo** (*) *[complex,private/allocatable]*
- updatewp_mod/**ddddw** (*,*) *[complex,private/allocatable]*
- updatewp_mod/**dif** (*) *[complex,private/allocatable]*
- updatewp_mod/**dwold** (*,*) *[complex,private/allocatable]*
- updatewp_mod/**ellmat_fd** *[type_tri_par,public]*
- updatewp_mod/**l_ellmat** (*) *[logical,private/allocatable]*
- updatewp_mod/**lwpmat** (*) *[logical,allocatable/public]*
- updatewp_mod/**maxthreads** *[integer,private]*
- updatewp_mod/**p0_ghost** (*) *[complex,allocatable/public]*
- updatewp_mod/**p0mat_fd** *[type_tri_par,public]*
- updatewp_mod/**pre** (*) *[complex,private/allocatable]*
- updatewp_mod/**rhs0** (*,*,*) *[real,private/allocatable]*
- updatewp_mod/**rhs1** (*,*,*) *[real,private/allocatable]*
- updatewp_mod/**size_rhs1** *[integer,private]*
- updatewp_mod/**w_ghost** (*,*) *[complex,allocatable/public]*
- updatewp_mod/**wmat_fd** *[type_penta_par,public]*
- updatewp_mod/**work** (*) *[real,private/allocatable]*

- updatewp_mod/**wpmat_fac** (*,*,*) *[real,private/allocatable]*

Subroutines and functions

subroutine updatewp_mod/**initialize_updatewp**(*tscheme*)

Purpose of this subroutine is to allocate the matrices needed to time advance the poloidal/pressure equations. Depending on the radial scheme, it can be either full or band matrices.

Parameters **tscheme** *[real]* :: time scheme

Called from *initialize_lmloop()*

subroutine updatewp_mod/**finalize_updatewp**(*tscheme*)

Deallocation of the matrices used to time-advance the poloidal/pressure equations.

Parameters **tscheme** *[real]* :: time scheme

Called from *finalize_lmloop()*

subroutine updatewp_mod/**updatewp**(*s*, *xi*, *w*, *dw*, *ddw*, *dwdt*, *p*, *dp*, *dpdt*, *tscheme*, *lrmsnext*, *lpressnext*)

updates the poloidal velocity potential w, the pressure p, and their radial derivatives.

Parameters

- **s** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **xi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **w** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dw** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **ddw** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dwdt** *[type_tarray,inout]*
- **p** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dp** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dpdt** *[type_tarray,inout]*
- **tscheme** *[real]*
- **lrmsnext** *[logical,in]*
- **lpressnext** *[logical,in]*

Called from *lmloop()*

Call to *get_p0mat()*, *get_wmat()*, *get_wpmat()*, *rint_r()*, *get_pol_rhs_imp()*

subroutine updatewp_mod/**preparew_fd**(*tscheme*, *dwdt*, *lpressnext*)

Parameters

- **tscheme** *[real]*
- **dwdt** *[type_tarray,inout]*

- **lpressnext** *[logical,in]*

Called from *test_lmloop()*, *lmloop_rdist()*

Call to *get_wmat_rdist()*, *get_p0mat_rdist()*, *get_openmp_blocks()*

subroutine updatewp_mod/**fill_ghosts_w**(*wg*, *p0g*, *lpressnext*)

This subroutine is used to fill the ghost zones.

Parameters

- **wg** (*lm_max*,5 - *nrstart* + *nrstop*) *[complex,inout]*
- **p0g** (3 - *nrstart* + *nrstop*) *[complex,inout]*
- **lpressnext** *[logical,in]*

Called from *lmloop_rdist()*, *getstartfields()*, *start_from_another_scheme()*, *assemble_pol_rloc()*

Call to *get_openmp_blocks()*

subroutine updatewp_mod/**updatew_fd**(*w*, *dw*, *ddw*, *dwdt*, *p*, *dp*, *dpdt*, *tscheme*, *lrmsnext*, *lpressnext*)

Parameters

- **w** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]* :: Poloidal potential
- **dw** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]* :: Radial derivative of w
- **ddw** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]* :: Radial derivative of dw
- **dwdt** *[type_tarray,inout]*
- **p** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]* :: Pressure
- **dp** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]* :: Radial derivative of p
- **dpdt** *[type_tarray,in]*
- **tscheme** *[real]*
- **lrmsnext** *[logical,in]*
- **lpressnext** *[logical,in]*

Called from *lmloop_rdist()*

Call to *get_pol_rhs_imp_ghost()*, *get_openmp_blocks()*

subroutine updatewp_mod/**get_pol**(*w*, *work*)

Get the poloidal potential from the solve of an elliptic equation. Careful: the output is in Chebyshev space!

Parameters

- **w** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **work** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*

Called from *assemble_pol()*

Call to *get_elliptic_mat()*

subroutine updatewp_mod/**finish_exp_pol**(*dvxvhlm*, *dw_exp_last*)

Parameters

- **dvxvhlm** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dw_exp_last** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*

Called from *finish_explicit_assembly()*

Call to *get_openmp_blocks()*

subroutine updatewp_mod/**finish_exp_pol_rdist**(*dvxvhlm*, *dw_exp_last*)

Parameters

- **dvxvhlm** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*
- **dw_exp_last** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*

Called from *finish_explicit_assembly_rdist()*

Call to *get_dr_rloc()*, *get_openmp_blocks()*

subroutine updatewp_mod/**get_pol_rhs_imp**(*s*, *xi*, *w*, *dw*, *ddw*, *p*, *dp*, *dwdt*, *dpdt*, *tscheme*, *istage*, *l_calc_lin*, *lpressnext*, *lrmsnext*, *dp_expl*[, *l_in_cheb_space*])

This subroutine computes the derivatives of w and p and assemble the implicit stage if needed.

Parameters

- **s** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **xi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **w** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dw** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **ddw** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **p** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dp** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dwdt** *[type_tarray,inout]*
- **dpdt** *[type_tarray,inout]*
- **tscheme** *[real]*
- **istage** *[integer,in]*
- **l_calc_lin** *[logical,in]*
- **lpressnext** *[logical,in]*
- **lrmsnext** *[logical,in]*
- **dp_expl** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **l_in_cheb_space** *[logical,in,]*

Called from *readstartfields_old()*, *readstartfields()*, *readstartfields_mpi()*, *getstartfields()*, *start_from_another_scheme()*, *updatewp()*

Call to *get_openmp_blocks()*, *get_ddr()*, *get_dddr()*, *hint2pol()*

subroutine updatewp_mod/**get_pol_rhs_imp_ghost**(*wg*, *dw*, *ddw*, *p*, *dp*, *dwdt*, *tscheme*, *istage*, *l_calc_lin*, *lpressnext*, *lrmsnext*, *dp_expl*)

This subroutine computes the derivatives of w and p and assemble the implicit stage if needed.

Parameters

- **wg** (*lm_max*,5 - *nrstart* + *nrstop*) *[complex,inout]*
- **dw** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **ddw** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **p** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*
- **dp** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **dwdt** *[type_tarray,inout]*
- **tscheme** *[real]*
- **istage** *[integer,in]*
- **l_calc_lin** *[logical,in]*
- **lpressnext** *[logical,in]*
- **lrmsnext** *[logical,in]*
- **dp_expl** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,in]*

Called from *getstartfields()*, *start_from_another_scheme()*, *updatew_fd()*, *assemble_pol_rloc()*

Call to *get_openmp_blocks()*, *get_ddddr_ghost()*, *get_dr_rloc()*, *hint2pol()*

subroutine updatewp_mod/**assemble_pol**(*s*, *xi*, *w*, *dw*, *ddw*, *p*, *dp*, *dwdt*, *dpdt*, *dp_expl*, *tscheme*, *lpressnext*, *lrmsnext*)

This subroutine is used to assemble w and dw/dr when IMEX RK time schemes which necessitate an assembly stage are employed. Robin-type boundary conditions are enforced using Canuto (1986) approach.

Parameters

- **s** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **xi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **w** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dw** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **ddw** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **p** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dp** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dwdt** *[type_tarray,inout]*
- **dpdt** *[type_tarray,inout]*
- **dp_expl** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **tscheme** *[real]*

- **lpressnext** *[logical,in]*
- **lrmsnext** *[logical,in]*

Called from *assemble_stage()*

Call to *get_pol()*, *get_openmp_blocks()*, *get_ddr()*, *hint2pol()*

subroutine updatewp_mod/**assemble_pol_rloc**(*block_sze*, *nblocks*, *w*, *dw*, *ddw*, *p*, *dp*, *dwdt*, *dp_expl*, *tscheme*, *lpressnext*, *lrmsnext*)

This subroutine is used to assemble w and dw/dr when IMEX RK time schemes which necessitate an assembly stage are employed. Robin-type boundary conditions are enforced using Canuto (1986) approach.

Parameters

- **block_sze** *[integer,in]*
- **nblocks** *[integer,in]*
- **w** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*
- **dw** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **ddw** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **p** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*
- **dp** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*
- **dwdt** *[type_tarray,inout]*
- **dp_expl** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,in]*
- **tscheme** *[real]*
- **lpressnext** *[logical,in]*
- **lrmsnext** *[logical,in]*

Called from *assemble_stage_rdist()*

Call to *get_elliptic_mat_rdist()*, *get_openmp_blocks()*, *bulk_to_ghost()*, *abortrun()*, *exch_ghosts()*, *fill_ghosts_w()*, *get_pol_rhs_imp_ghost()*

subroutine updatewp_mod/**get_wpmat**(*tscheme*, *l*, *hdif*, *wpmat*, *wpmat_fac*)

Purpose of this subroutine is to contruct the time step matrix `wpMat` for the Navier-Stokes equation.

Parameters

- **tscheme** *[real]* :: time scheme
- **l** *[integer,in]* :: degree ℓ
- **hdif** *[real,in]* :: hyperdiffusion
- **wpmat** *[real]* :: ‘)
- **wpmat_fac** (2 * *n_r_max*,2) *[real,out]*

Called from *updatewp()*

Call to *abortrun()*

subroutine updatewp_mod/**get_elliptic_mat**(*l*, *ellmat*)

Purpose of this subroutine is to contruct the matrix needed for the derivation of w for the time advance of the poloidal equation if the double curl form is used.

Parameters

- **l** *[integer,in]* :: degree ℓ
- **ellmat** *[real]* :: ‘)

Called from *get_pol()*

Call to *abortrun()*

subroutine updatewp_mod/**get_wmat**(*tscheme*, *l*, *hdif*, *wmat*, *wmat_fac*)

Purpose of this subroutine is to contruct the time step matrix wpmat for the NS equation. This matrix corresponds here to the radial component of the double-curl of the Navier-Stokes equation.

Parameters

- **tscheme** *[real]* :: time scheme
- **l** *[integer,in]* :: degree ℓ
- **hdif** *[real,in]* :: hyperdiffusion
- **wmat** *[real]* :: ‘)
- **wmat_fac** (*n_r_max*,2) *[real,out]*

Called from *updatewp()*

Call to *abortrun()*

subroutine updatewp_mod/**get_elliptic_mat_rdist**(*ellmat*)

Purpose of this subroutine is to contruct the matrix needed for the derivation of w for the time advance of the poloidal equation if the double curl form is used. This is the R-dist version.

Parameters **ellmat** *[type_tri_par,inout]*

Called from *assemble_pol_rloc()*

subroutine updatewp_mod/**get_wmat_rdist**(*tscheme*, *hdif*, *wmat*)

Purpose of this subroutine is to contruct the time step matrix wMat_FD for the NS equation. This matrix corresponds here to the radial component of the double-curl of the Navier-Stokes equation. This routine is used when parallel F.D. solvers are employed.

Parameters

- **tscheme** *[real]* :: time scheme
- **hdif** (1 + *l_max*) *[real,in]* :: hyperdiffusion
- **wmat** *[type_penta_par,inout]*

Called from *preparew_fd()*

subroutine updatewp_mod/**get_p0mat**(*pmat*)

This subroutine solves the linear problem of the spherically-symmetric pressure

Parameters **pmat** *[real]* :: matrix

Called from *updatewp()*

Call to *abortrun()*

subroutine updatewp_mod/**get_p0mat_rdist**(*pmat*)

This subroutine solves the linear problem of the spherically-symmetric pressure. This is the R-distributed variant of the function used when parallel F.D. solvers are employed

Parameters **pmat** *[type_tri_par,inout]* :: matrix

Called from *preparew_fd()*

10.8.4 updateZ.f90

Description

This module handles the time advance of the toroidal potential z It contains the computation of the implicit terms and the linear solves.

Quick access

Variables *dif*, *lz10mat*, *lzmat*, *maxthreads*, *rhs1*, *z10_ghost*, *z10mat_fac*, *z10mat_fd*, *z_ghost*, *zmat_fac*, *zmat_fd*

Routines *assemble_tor()*, *assemble_tor_rloc()*, *fill_ghosts_z()*, *finalize_updatez()*, *finish_exp_tor()*, *get_tor_rhs_imp()*, *get_tor_rhs_imp_ghost()*, *get_z10mat()*, *get_z10mat_rdist()*, *get_zmat()*, *get_zmat_rdist()*, *initialize_updatez()*, *preparez_fd()*, *update_rot_rates()*, *update_rot_rates_rloc()*, *updatez()*, *updatez_fd()*

Needed modules

- *init_fields*: This module is used to construct the initial solution.
- omp_lib
- *precision_mod*: This module controls the precision used in MagIC
- *communications* (*allgather_from_rloc()*): This module contains the different MPI communicators used in MagIC.
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *truncation* (*n_r_max()*, *lm_max()*, *l_max()*): This module defines the grid points and the truncation
- *radial_data* (*n_r_cmb()*, *n_r_icb()*, *nrstart()*, *nrstop()*): This module defines the MPI decomposition in the radial direction.
- *radial_functions* (*visc()*, *or1()*, *or2()*, rscheme_oc(), *dlvisc()*, *beta()*, *rho0()*, *r_icb()*, *r_cmb()*, *r()*, *dbeta()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)

- *physical_parameters* (*kbotv()*, *ktopv()*, *prec_angle()*, *po()*, *oek()*): Module containing the physical parameters
- *num_param* (*amstart()*, *dct_counter()*, *solve_counter()*): Module containing numerical and control parameters
- *torsional_oscillations* (*ddzasl()*): This module contains information for TO calculation and output
- *blocking* (*lo_sub_map()*, *lo_map()*, *st_sub_map()*, *llm()*, *ulm()*, *st_map()*): Module containing blocking information
- *horizontal_data* (*hdif_v()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *logic* (*l_rot_ma()*, *l_rot_ic()*, *l_srma()*, *l_sric()*, *l_z10mat()*, *l_precession()*, *l_correct_ame()*, *l_correct_amz()*, *l_update_v()*, *l_to()*, *l_finite_diff()*, *l_full_sphere()*, *l_parallel_solve()*): Module containing the logicals that control the run
- *rms* (*diftor2hint()*): This module contains the calculation of the RMS force balance and induction terms.
- *constants* (*c_lorentz_ma()*, *c_lorentz_ic()*, *c_dt_z10_ma()*, *c_dt_z10_ic()*, *c_moi_ma()*, *c_moi_ic()*, *c_z10_omega_ma()*, *c_z10_omega_ic()*, *c_moi_oc()*, *y10_norm()*, *y11_norm()*, *zero()*, *one()*, *two()*, *four()*, *pi()*, *third()*): module containing constants and parameters used in the code.
- *parallel_mod*: This module contains the blocking information
- *outrot* (*get_angular_moment()*, *get_angular_moment_rloc()*): This module handles the writing of several diagnostic files related to the rotation: angular momentum (AM.TAG), drift (drift.TAG), inner core and mantle rotations....
- *rms_helpers* (*hint2tor()*): This module contains several useful subroutines required to compute RMS diagnostics
- *radial_der* (*get_ddr()*, *get_ddr_ghost()*, *bulk_to_ghost()*, *exch_ghosts()*): Radial derivatives functions
- *fields* (*work_lmloc()*): This module contains all the fields used in MagIC in the hybrid (LM,r) space as well as their radial derivatives. It defines both the LM-distributed arrays and the R-distributed arrays....
- *useful* (*abortrun()*): This module contains several useful routines.
- *time_schemes* (`type_tscheme()`): This module defines an abstract class type_tscheme which is employed for the time advance of the code.
- *time_array* (`type_tarray()`, `type_tscalar()`): This module defines two types that are defined to store the implicit/explicit terms at the different sub-stage/step.
- *special*: This module contains all variables for the case of an imposed IC dipole, an imposed external magnetic field and a special boundary forcing to excite inertial modes
- *dense_matrices*
- *real_matrices*
- *band_matrices*
- *parallel_solvers*: This module contains the routines that are used to solve linear banded problems with R-distributed arrays.

Variables

- updatez_mod/**dif** (*) *[complex,private/allocatable]*
- updatez_mod/**lz10mat** *[logical,public]*
- updatez_mod/**lzmat** (*) *[logical,allocatable/public]*
- updatez_mod/**maxthreads** *[integer,private]*
- updatez_mod/**rhs1** (*,*,*) *[real,private/allocatable]*
  RHS for other modes
- updatez_mod/**z10_ghost** (*) *[complex,allocatable/public]*
- updatez_mod/**z10mat_fac** (*) *[real,private/allocatable]*
- updatez_mod/**z10mat_fd** *[type_tri_par,public]*
- updatez_mod/**z_ghost** (*,*) *[complex,allocatable/public]*
- updatez_mod/**zmat_fac** (*,*) *[real,private/allocatable]*
- updatez_mod/**zmat_fd** *[type_tri_par,public]*

Subroutines and functions

subroutine updatez_mod/**initialize_updatez**()
This subroutine handles the memory allocation of the arrays involved in the time advance of the toroidal potential.

Called from *initialize_lmloop()*

subroutine updatez_mod/**finalize_updatez**()
Memory deallocation of arrays associated with time advance of Z

Called from *finalize_lmloop()*

subroutine updatez_mod/**updatez**(*time*, *timenext*, *z*, *dz*, *dzdt*, *omega_ma*, *omega_ic*, *domega_ma_dt*, *domega_ic_dt*, *lorentz_torque_ma_dt*, *lorentz_torque_ic_dt*, *tscheme*, *lrmsnext*)

Updates the toroidal potential z and its radial derivative dz

Parameters

- **time** *[real,in]* :: Current stage time
- **timenext** *[real,in]* :: Next time
- **z** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]* :: Toroidal velocity potential z
- **dz** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]* :: Radial derivative of z
- **dzdt** *[type_tarray,inout]*
- **omega_ma** *[real,out]* :: Calculated OC rotation
- **omega_ic** *[real,out]* :: Calculated IC rotation
- **domega_ma_dt** *[type_tscalar,inout]*
- **domega_ic_dt** *[type_tscalar,inout]*
- **lorentz_torque_ma_dt** *[type_tscalar,inout]*
- **lorentz_torque_ic_dt** *[type_tscalar,inout]*

- **tscheme** *[real]*
- **lrmsnext** *[logical,in]* :: Logical for storing update if (l_RMS.and.l_logNext)

Called from *lmloop()*

Call to *get_zmat()*, *get_z10mat()*, *update_rot_rates()*, *get_tor_rhs_imp()*

subroutine updatez_mod/**preparez_fd**(*time*, *tscheme*, *dzdt*, *omega_ma*, *omega_ic*, *domega_ma_dt*, *domega_ic_dt*)

– Input of variables:

Parameters

- **time** *[real,in]*
- **tscheme** *[real]*
- **dzdt** *[type_tarray,inout]*
- **omega_ma** *[real,inout]* :: Calculated OC rotation
- **omega_ic** *[real,inout]* :: Calculated IC rotation
- **domega_ma_dt** *[type_tscalar,inout]*
- **domega_ic_dt** *[type_tscalar,inout]*

Called from *test_lmloop()*, *lmloop_rdist()*

Call to *get_zmat_rdist()*, *get_z10mat_rdist()*, *get_openmp_blocks()*

subroutine updatez_mod/**fill_ghosts_z**(*zg*)

This subroutine is used to fill the ghosts zones that are located at nR=n_r_cmb-1 and nR=n_r_icb+1. This is used to properly set the Neuman boundary conditions. In case Dirichlet BCs are used, a simple first order extrapolation is employed. This is anyway only used for outputs (like Nusselt numbers).

Parameters **zg** (*lm_max*,3 - *nrstart* + *nrstop*) *[complex,inout]*

Called from *lmloop_rdist()*, *getstartfields()*, *start_from_another_scheme()*, *assemble_tor_rloc()*

Call to *get_openmp_blocks()*

subroutine updatez_mod/**updatez_fd**(*time*, *timenext*, *z*, *dz*, *dzdt*, *omega_ma*, *omega_ic*, *domega_ma_dt*, *domega_ic_dt*, *lorentz_torque_ma_dt*, *lorentz_torque_ic_dt*, *tscheme*, *lrmsnext*)

– Input of variables:

Parameters

- **time** *[real,in]* :: Current stage time
- **timenext** *[real,in]* :: Next time
- **z** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]* :: Toroidal potential
- **dz** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]* :: Radial derivative of z
- **dzdt** *[type_tarray,inout]*
- **omega_ma** *[real,inout]* :: Calculated OC rotation
- **omega_ic** *[real,inout]* :: Calculated IC rotation

- **domega_ma_dt** *[type_tscalar,inout]*
- **domega_ic_dt** *[type_tscalar,inout]*
- **lorentz_torque_ma_dt** *[type_tscalar,inout]*
- **lorentz_torque_ic_dt** *[type_tscalar,inout]*
- **tscheme** *[real]*
- **lrmsnext** *[logical,in]* :: Logical for storing update if (l_RMS.and.l_logNext)

Called from *lmloop_rdist()*

Call to *update_rot_rates_rloc()*, *get_tor_rhs_imp_ghost()*, *get_openmp_blocks()*

subroutine updatez_mod/**get_tor_rhs_imp**(*time*, *z*, *dz*, *dzdt*, *domega_ma_dt*, *domega_ic_dt*, *omega_ic*, *omega_ma*, *omega_ic1*, *omega_ma1*, *tscheme*, *istage*, *l_calc_lin*, *lrmsnext*[, *l_in_cheb_space*])

– Input variables

Parameters

- **time** *[real,in]*
- **z** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dz** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dzdt** *[type_tarray,inout]*
- **domega_ma_dt** *[type_tscalar,inout]*
- **domega_ic_dt** *[type_tscalar,inout]*
- **omega_ic** *[real,inout]*
- **omega_ma** *[real,inout]*
- **omega_ic1** *[real,inout]*
- **omega_ma1** *[real,inout]*
- **tscheme** *[real]*
- **istage** *[integer,in]*
- **l_calc_lin** *[logical,in]*
- **lrmsnext** *[logical,in]*
- **l_in_cheb_space** *[logical,in,]*

Called from *readstartfields_old()*, *readstartfields()*, *readstartfields_mpi()*, *getstartfields()*, *start_from_another_scheme()*, *updatez()*, *assemble_tor()*

Call to *get_openmp_blocks()*, *get_ddr()*, *get_angular_moment()*, *hint2tor()*

subroutine updatez_mod/**get_tor_rhs_imp_ghost**(*time*, *zg*, *dz*, *dzdt*, *domega_ma_dt*, *domega_ic_dt*, *omega_ic*, *omega_ma*, *omega_ic1*, *omega_ma1*, *tscheme*, *istage*, *l_calc_lin*, *lrmsnext*)

– Input variables

Parameters

- **time** *[real,in]*
- **zg** (*lm_max*,3 - *nrstart* + *nrstop*) *[complex,inout]*

- **dz** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **dzdt** *[type_tarray,inout]*
- **domega_ma_dt** *[type_tscalar,inout]*
- **domega_ic_dt** *[type_tscalar,inout]*
- **omega_ic** *[real,inout]*
- **omega_ma** *[real,inout]*
- **omega_ic1** *[real,inout]*
- **omega_ma1** *[real,inout]*
- **tscheme** *[real]*
- **istage** *[integer,in]*
- **l_calc_lin** *[logical,in]*
- **lrmsnext** *[logical,in]*

Called from *getstartfields()*, *start_from_another_scheme()*, *updatez_fd()*, *assemble_tor_rloc()*

Call to *get_openmp_blocks()*, *get_ddr_ghost()*, *get_angular_moment_rloc()*, *hint2tor()*

subroutine updatez_mod/**assemble_tor**(*time*, *z*, *dz*, *dzdt*, *domega_ic_dt*, *domega_ma_dt*, *lorentz_torque_ic_dt*, *lorentz_torque_ma_dt*, *omega_ic*, *omega_ma*, *omega_ic1*, *omega_ma1*, *lrmsnext*, *tscheme*)

This subroutine is used to assemble the toroidal flow potential when an IMEX RK time scheme with an assembly stage is employed (LM-distributed version).

Parameters

- **time** *[real,in]*
- **z** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dz** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dzdt** *[type_tarray,inout]*
- **domega_ic_dt** *[type_tscalar,inout]*
- **domega_ma_dt** *[type_tscalar,inout]*
- **lorentz_torque_ic_dt** *[type_tscalar,inout]*
- **lorentz_torque_ma_dt** *[type_tscalar,inout]*
- **omega_ic** *[real,inout]*
- **omega_ma** *[real,inout]*
- **omega_ic1** *[real,inout]*
- **omega_ma1** *[real,inout]*
- **lrmsnext** *[logical,in]*
- **tscheme** *[real]*

Called from *assemble_stage()*

Call to *abortrun()*, *update_rot_rates()*, *get_tor_rhs_imp()*

subroutine updatez_mod/**assemble_tor_rloc**(*time*, *z*, *dz*, *dzdt*, *domega_ic_dt*, *domega_ma_dt*, *lorentz_torque_ic_dt*, *lorentz_torque_ma_dt*, *omega_ic*, *omega_ma*, *omega_ic1*, *omega_ma1*, *lrmsnext*, *tscheme*)

This subroutine is used when a IMEX Runge-Kutta scheme with an assembly stage is employed (R-distributed version)

Parameters

- **time** *[real,in]*
- **z** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*
- **dz** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **dzdt** *[type_tarray,inout]*
- **domega_ic_dt** *[type_tscalar,inout]*
- **domega_ma_dt** *[type_tscalar,inout]*
- **lorentz_torque_ic_dt** *[type_tscalar,inout]*
- **lorentz_torque_ma_dt** *[type_tscalar,inout]*
- **omega_ic** *[real,inout]*
- **omega_ma** *[real,inout]*
- **omega_ic1** *[real,inout]*
- **omega_ma1** *[real,inout]*
- **lrmsnext** *[logical,in]*
- **tscheme** *[real]*

Called from *assemble_stage_rdist()*

Call to *abortrun()*, *get_openmp_blocks()*, *bulk_to_ghost()*, *exch_ghosts()*, *fill_ghosts_z()*, *update_rot_rates_rloc()*, *get_tor_rhs_imp_ghost()*

subroutine updatez_mod/**update_rot_rates**(*z*, *lo_ma*, *lo_ic*, *lorentz_torque_ma_dt*, *lorentz_torque_ic_dt*, *omega_ma*, *omega_ma1*, *omega_ic*, *omega_ic1*, *istage*[, *l_in_cheb_space*])

This subroutine updates the rotation rate of inner core and mantle.

Parameters

- **z** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **lo_ma** *[real,in]*
- **lo_ic** *[real,in]* :: RHS when stress-free BCs are used
- **lorentz_torque_ma_dt** *[type_tscalar,inout]*
- **lorentz_torque_ic_dt** *[type_tscalar,inout]*
- **omega_ma** *[real,out]*
- **omega_ma1** *[real,out]*

- **omega_ic** *[real,out]*
- **omega_ic1** *[real,out]*
- **istage** *[integer,in]*
- **l_in_cheb_space** *[logical,in,]* :: Is z in Cheb space or not?

Called from *updatez()*, *assemble_tor()*

subroutine updatez_mod/**update_rot_rates_rloc**(*z*, *lo_ma*, *lo_ic*, *lorentz_torque_ma_dt*, *lorentz_torque_ic_dt*, *omega_ma*, *omega_ma1*, *omega_ic*, *omega_ic1*, *istage*)

– Input variables

Parameters

- **z** (*lm_max*,3 - *nrstart* + *nrstop*) *[complex,in]*
- **lo_ma** *[real,in]*
- **lo_ic** *[real,in]* :: RHS when stress-free BCs are used
- **lorentz_torque_ma_dt** *[type_tscalar,inout]*
- **lorentz_torque_ic_dt** *[type_tscalar,inout]*
- **omega_ma** *[real,out]*
- **omega_ma1** *[real,out]*
- **omega_ic** *[real,out]*
- **omega_ic1** *[real,out]*
- **istage** *[integer,in]*

Called from *updatez_fd()*, *assemble_tor_rloc()*

subroutine updatez_mod/**finish_exp_tor**(*lorentz_torque_ma*, *lorentz_torque_ic*, *domega_ma_dt_exp*, *domega_ic_dt_exp*, *lorentz_ma_exp*, *lorentz_ic_exp*)

– Input variables

Parameters

- **lorentz_torque_ma** *[real,in]* :: Lorentz torque (for OC rotation)
- **lorentz_torque_ic** *[real,in]* :: Lorentz torque (for IC rotation)
- **domega_ma_dt_exp** *[real,out]*
- **domega_ic_dt_exp** *[real,out]*
- **lorentz_ma_exp** *[real,out]*
- **lorentz_ic_exp** *[real,out]*

Called from *finish_explicit_assembly()*, *finish_explicit_assembly_rdist()*

subroutine updatez_mod/**get_z10mat**(*tscheme*, *l*, *hdif*, *zmat*, *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 $\ell = 1$ 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

- **tscheme** *[real]* :: Time step internal
- **l** *[integer,in]* :: Variable to loop over degrees
- **hdif** *[real,in]* :: Value of hyperdiffusivity in zMat terms
- **zmat** *[real]*
- **zmat_fac** (*n_r_max*) *[real,out]* :: Inverse of max(zMat) for inversion

Called from *updatez()*

Call to *abortrun()*

subroutine updatez_mod/**get_zmat**(*tscheme*, *l*, *hdif*, *zmat*, *zmat_fac*)

Purpose of this subroutine is to contruct the time step matrices `zmat(i,j)` for the Navier-Stokes equation.

Parameters

- **tscheme** *[real]* :: time step
- **l** *[integer,in]* :: Variable to loop over degrees
- **hdif** *[real,in]* :: Hyperdiffusivity
- **zmat** *[real]*
- **zmat_fac** (*n_r_max*) *[real,out]* :: Inverse of max(zMat) for the inversion

Called from *updatez()*

Call to *abortrun()*

subroutine updatez_mod/**get_z10mat_rdist**(*tscheme*, *hdif*, *zmat*)

This subroutine is employed to construct the matrix for the z(l=1,m=0) mode when the parallel F.D. solver is used.

Parameters

- **tscheme** *[real]* :: Time step internal
- **hdif** (1 + *l_max*) *[real,in]* :: Value of hyperdiffusivity in zMat terms
- **zmat** *[type_tri_par,inout]* :: Tridiag matrix

Called from *preparez_fd()*

Call to *abortrun()*

subroutine updatez_mod/**get_zmat_rdist**(*tscheme*, *hdif*, *zmat*)

This subroutine is used to construct the z matrices when the parallel F.D. solver is employed.

Parameters

- **tscheme** *[real]* :: time step
- **hdif** (1 + *l_max*) *[real,in]* :: Hyperdiffusivity
- **zmat** *[type_tri_par,inout]*

Called from *preparez_fd()*

10.8.5 updateS.f90

Description

This module handles the time advance of the entropy s. It contains the computation of the implicit terms and the linear solves.

Quick access

Variables *lsmat*, *s0mat_fac*, *s_ghost*, *smat_fac*, *smat_fd*

Routines *assemble_entropy()*, *assemble_entropy_rloc()*, *fill_ghosts_s()*, *finalize_updates()*, *finish_exp_entropy()*, *finish_exp_entropy_rdist()*, *get_entropy_rhs_imp()*, *get_entropy_rhs_imp_ghost()*, *get_s0mat()*, *get_smat()*, *get_smat_rdist()*, *initialize_updates()*, *prepares_fd()*, *updates()*, *updates_fd()*

Needed modules

- omp_lib
- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *truncation* (*n_r_max()*, *lm_max()*, *l_max()*): This module defines the grid points and the truncation
- *radial_data* (*n_r_cmb()*, *n_r_icb()*, *nrstart()*, *nrstop()*): This module defines the MPI decomposition in the radial direction.
- *radial_functions* (*orho1()*, *or1()*, *or2()*, *beta()*, *dentropy0()*, rscheme_oc(), *kappa()*, *dlkappa()*, *dltemp0()*, *temp0()*, *r()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *physical_parameters* (*opr()*, *kbots()*, *ktops()*, *stef()*): Module containing the physical parameters
- *num_param* (*dct_counter()*, *solve_counter()*): Module containing numerical and control parameters
- *init_fields* (*tops()*, *bots()*): This module is used to construct the initial solution.
- *blocking* (*lo_map()*, *lo_sub_map()*, *llm()*, *ulm()*, *st_map()*): Module containing blocking information
- *horizontal_data* (*hdif_s()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *logic* (*l_update_s()*, *l_anelastic_liquid()*, *l_finite_diff()*, *l_full_sphere()*, *l_parallel_solve()*, *l_phase_field()*): Module containing the logicals that control the run
- *parallel_mod*: This module contains the blocking information
- *radial_der* (*get_ddr()*, *get_dr()*, *get_dr_rloc()*, *get_ddr_ghost()*, *exch_ghosts()*, *bulk_to_ghost()*): Radial derivatives functions
- *fields* (*work_lmloc()*): This module contains all the fields used in MagIC in the hybrid (LM,r) space as well as their radial derivatives. It defines both the LM-distributed arrays and the R-distributed arrays....
- *constants* (*zero()*, *one()*, *two()*): module containing constants and parameters used in the code.
- *useful* (*abortrun()*): This module contains several useful routines.

- *time_schemes* (type_tscheme()): This module defines an abstract class type_tscheme which is employed for the time advance of the code.
- *time_array* (type_tarray()): This module defines two types that are defined to store the implicit/explicit terms at the different sub-stage/step.
- *dense_matrices*
- *real_matrices*
- *band_matrices*
- *parallel_solvers* (type_tri_par()): This module contains the routines that are used to solve linear banded problems with R-distributed arrays.

Variables

- updates_mod/**fd_fac_bot** (*) *[real,private/allocatable]*
- updates_mod/**fd_fac_top** (*) *[real,private/allocatable]*
- updates_mod/**lsmat** (*) *[logical,allocatable/public]*
- updates_mod/**maxthreads** *[integer,private]*
- updates_mod/**rhs1** (*,*,*) *[real,private/allocatable]*
- updates_mod/**s0mat_fac** (*) *[real,private/allocatable]*
- updates_mod/**s_ghost** (*,*) *[complex,public/allocatable]*
- updates_mod/**smat_fac** (*,*) *[real,private/allocatable]*
- updates_mod/**smat_fd** *[type_tri_par,public]*

Subroutines and functions

subroutine updates_mod/**initialize_updates**()

This subroutine allocates the arrays involved in the time-advance of the entropy/temperature equation.

Called from *initialize_lmloop()*

subroutine updates_mod/**finalize_updates**()

Memory deallocation of updateS module

Called from *finalize_lmloop()*

subroutine updates_mod/**updates**(*s*, *ds*, *dsdt*, *phi*, *tscheme*)

Updates the entropy field s and its radial derivative.

Parameters

- **s** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]* :: Entropy
- **ds** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]* :: Radial derivative of entropy
- **dsdt** *[type_tarray,inout]*
- **phi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]* :: Phase field
- **tscheme** *[real]*

Called from *lmloop()*

Call to *get_s0mat()*, *get_smat()*, *get_entropy_rhs_imp()*

subroutine updates_mod/**prepares_fd**(*tscheme*, *dsdt*, *phi*)

This subroutine is used to assemble the r.h.s. of the entropy equation when parallel F.D solvers are used. Boundary values are set here.

Parameters

- **tscheme** *[real]*
- **dsdt** *[type_tarray,inout]*
- **phi** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,in]*

Called from *test_lmloop()*, *lmloop_rdist()*

Call to *get_smat_rdist()*, *get_openmp_blocks()*

subroutine updates_mod/**fill_ghosts_s**(*sg*)

This subroutine is used to fill the ghosts zones that are located at nR=n_r_cmb-1 and nR=n_r_icb+1. This is used to properly set the Neuman boundary conditions. In case Dirichlet BCs are used, a simple first order extrapolation is employed. This is anyway only used for outputs (like Nusselt numbers).

Parameters **sg** (*lm_max*,3 - *nrstart* + *nrstop*) *[complex,inout]*

Called from *lmloop_rdist()*, *getstartfields()*, *start_from_another_scheme()*, *assemble_entropy_rloc()*

Call to *get_openmp_blocks()*

subroutine updates_mod/**updates_fd**(*s*, *ds*, *dsdt*, *phi*, *tscheme*)

This subroutine is called after the linear solves have been completed. This is then assembling the linear terms that will be used in the r.h.s. for the next iteration.

Parameters

- **s** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]* :: Entropy
- **ds** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]* :: Radial derivative of entropy
- **dsdt** *[type_tarray,inout]*
- **phi** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,in]* :: Phase field
- **tscheme** *[real]*

Called from *lmloop_rdist()*

Call to *get_entropy_rhs_imp_ghost()*, *get_openmp_blocks()*

subroutine updates_mod/**finish_exp_entropy**(*w*, *dvsrlm*, *ds_exp_last*)

This subroutine completes the computation of the advection term by computing the radial derivative (LM-distributed variant).

Parameters

- **w** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*

- **dvsrlm** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **ds_exp_last** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*

Called from *finish_explicit_assembly()*

Call to *get_openmp_blocks()*

subroutine updates_mod/**finish_exp_entropy_rdist**(*w*, *dvsrlm*, *ds_exp_last*)

This subroutine completes the computation of the advection term by computing the radial derivative (R-distributed variant).

Parameters

- **w** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,in]*
- **dvsrlm** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*
- **ds_exp_last** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*

Called from *finish_explicit_assembly_rdist()*

Call to *get_dr_rloc()*, *get_openmp_blocks()*

subroutine updates_mod/**get_entropy_rhs_imp**(*s*, *ds*, *dsdt*, *phi*, *istage*, *l_calc_lin*[, *l_in_cheb_space*])

This subroutine computes the linear terms that enters the r.h.s.. This is used with LM-distributed

Parameters

- **s** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **ds** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dsdt** *[type_tarray,inout]*
- **phi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **istage** *[integer,in]*
- **l_calc_lin** *[logical,in]*
- **l_in_cheb_space** *[logical,in,]*

Called from *readstartfields_old()*, *readstartfields()*, *readstartfields_mpi()*, *getstartfields()*, *start_from_another_scheme()*, *updates()*, *assemble_entropy()*

Call to *get_openmp_blocks()*, *get_ddr()*

subroutine updates_mod/**get_entropy_rhs_imp_ghost**(*sg*, *ds*, *dsdt*, *phi*, *istage*, *l_calc_lin*)

This subroutine computes the linear terms that enters the r.h.s.. This is used with R-distributed

Parameters

- **sg** (*lm_max*,3 - *nrstart* + *nrstop*) *[complex,in]*
- **ds** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **dsdt** *[type_tarray,inout]*
- **phi** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,in]*
- **istage** *[integer,in]*

- **l_calc_lin** *[logical,in]*

Called from *getstartfields()*, *start_from_another_scheme()*, *updates_fd()*, *assemble_entropy_rloc()*

Call to *get_openmp_blocks()*, *get_ddr_ghost()*

subroutine updates_mod/**assemble_entropy_rloc**(*s*, *ds*, *dsdt*, *phi*, *tscheme*)

This subroutine is used when an IMEX Runge-Kutta time scheme with an assembly stage is used. This is used when R is distributed.

Parameters

- **s** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*
- **ds** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **dsdt** *[type_tarray,inout]*
- **phi** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,in]*
- **tscheme** *[real]*

Called from *assemble_stage_rdist()*

Call to *get_openmp_blocks()*, *bulk_to_ghost()*, *exch_ghosts()*, *fill_ghosts_s()*, *get_entropy_rhs_imp_ghost()*

subroutine updates_mod/**assemble_entropy**(*s*, *ds*, *dsdt*, *phi*, *tscheme*)

This subroutine is used to assemble the entropy/temperature at assembly stages of IMEX-RK time schemes. This is used when LM is distributed.

Parameters

- **s** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **ds** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dsdt** *[type_tarray,inout]*
- **phi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **tscheme** *[real]*

Called from *assemble_stage()*

Call to *get_entropy_rhs_imp()*

subroutine updates_mod/**get_s0mat**(*tscheme*, *smat*, *smat_fac*)

Purpose of this subroutine is to contruct the time step matrix sMat0

Parameters

- **tscheme** *[real]* :: time step
- **smat** *[real]*
- **smat_fac** (*n_r_max*) *[real,out]*

Called from *updates()*

Call to *abortrun()*

subroutine updates_mod/**get_smat**(*tscheme*, *l*, *hdif*, *smat*, *smat_fac*)

Purpose of this subroutine is to contruct the time step matrices sMat(i,j) and s0mat for the entropy equation.

Parameters

- **tscheme** *[real]* :: time step
- **l** *[integer,in]*
- **hdif** *[real,in]*
- **smat** *[real]* :: ‘)
- **smat_fac** (*n_r_max*) *[real,out]*

Called from *updates()*

Call to *abortrun()*

subroutine updates_mod/**get_smat_rdist**(*tscheme*, *hdif*, *smat*)

This subroutine is used to construct the matrices when the parallel solver for F.D. is employed.

Parameters

- **tscheme** *[real]* :: time step
- **hdif** (1 + *l_max*) *[real,in]*
- **smat** *[type_tri_par,inout]*

Called from *prepares_fd()*

10.8.6 updateXI.f90

Description

This module handles the time advance of the chemical composition xi. It contains the computation of the implicit terms and the linear solves.

Quick access

Variables *fd_fac_bot*, *fd_fac_top*, *lximat*, *xi0mat_fac*, *xi_ghost*, *ximat_fac*, *ximat_fd*

Routines *assemble_comp()*, *assemble_comp_rloc()*, *fill_ghosts_xi()*, *finalize_updatexi()*, *finish_exp_comp()*, *finish_exp_comp_rdist()*, *get_comp_rhs_imp()*, *get_comp_rhs_imp_ghost()*, *get_xi0mat()*, *get_ximat()*, *get_ximat_rdist()*, *initialize_updatexi()*, *preparexi_fd()*, *updatexi()*, *updatexi_fd()*

Needed modules

- omp_lib
- *precision_mod*: This module controls the precision used in MagIC
- *truncation* (*n_r_max()*, *lm_max()*, *l_max()*): This module defines the grid points and the truncation
- *radial_data* (*n_r_icb()*, *n_r_cmb()*, *nrstart()*, *nrstop()*): This module defines the MPI decomposition in the radial direction.
- *radial_functions* (*orho1()*, *or1()*, *or2()*, *beta()*, rscheme_oc(), *r()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *physical_parameters* (*osc()*, *kbotxi()*, *ktopxi()*): Module containing the physical parameters
- *num_param* (*dct_counter()*, *solve_counter()*): Module containing numerical and control parameters
- *init_fields* (*topxi()*, *botxi()*): This module is used to construct the initial solution.
- *blocking* (*lo_map()*, *lo_sub_map()*, *llm()*, *ulm()*, *st_map()*): Module containing blocking information
- *horizontal_data* (*hdif_xi()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *logic* (*l_update_xi()*, *l_finite_diff()*, *l_full_sphere()*, *l_parallel_solve()*): Module containing the logicals that control the run
- *parallel_mod* (rank(), *chunksize()*, *n_procs()*, *get_openmp_blocks()*): This module contains the blocking information
- *radial_der* (*get_ddr()*, *get_dr()*, *get_dr_rloc()*, *get_ddr_ghost()*, *exch_ghosts()*, *bulk_to_ghost()*): Radial derivatives functions
- *constants* (*zero()*, *one()*, *two()*): module containing constants and parameters used in the code.
- *fields* (*work_lmloc()*): This module contains all the fields used in MagIC in the hybrid (LM,r) space as well as their radial derivatives. It defines both the LM-distributed arrays and the R-distributed arrays. ...
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *useful* (*abortrun()*): This module contains several useful routines.
- *time_schemes* (type_tscheme()): This module defines an abstract class type_tscheme which is employed for the time advance of the code.
- *time_array* (type_tarray()): This module defines two types that are defined to store the implicit/explicit terms at the different sub-stage/step.
- *dense_matrices*
- *real_matrices*
- *band_matrices*
- *parallel_solvers* (type_tri_par()): This module contains the routines that are used to solve linear banded problems with R-distributed arrays.

Variables

- updatexi_mod/**fd_fac_bot** (*) *[complex,private/allocatable]*
- updatexi_mod/**fd_fac_top** (*) *[complex,private/allocatable]*
- updatexi_mod/**lximat** (*) *[logical,allocatable/public]*
- updatexi_mod/**maxthreads** *[integer,private]*
- updatexi_mod/**rhs1** (*,*,*) *[real,private/allocatable]*
- updatexi_mod/**xi0mat_fac** (*) *[real,private/allocatable]*
- updatexi_mod/**xi_ghost** (*,*) *[complex,allocatable/public]*
- updatexi_mod/**ximat_fac** (*,*) *[real,private/allocatable]*
- updatexi_mod/**ximat_fd** *[type_tri_par,public]*

Subroutines and functions

subroutine updatexi_mod/**initialize_updatexi**()

Called from *initialize_lmloop()*

subroutine updatexi_mod/**finalize_updatexi**()

This subroutine deallocates the matrices involved in the time-advance of xi.

Called from *finalize_lmloop()*

subroutine updatexi_mod/**updatexi**(*xi*, *dxi*, *dxidt*, *tscheme*)

Updates the chemical composition field s and its radial derivative.

Parameters

- **xi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]* :: Chemical composition
- **dxi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]* :: Radial derivative of xi
- **dxidt** *[type_tarray,inout]*
- **tscheme** *[real]*

Called from *lmloop()*

Call to *get_xi0mat()*, *get_ximat()*, *get_comp_rhs_imp()*

subroutine updatexi_mod/**preparexi_fd**(*tscheme*, *dxidt*)

This subroutine is used to assemble the r.h.s. of the composition equation when parallel F.D solvers are used. Boundary values are set here.

Parameters

- **tscheme** *[real]*
- **dxidt** *[type_tarray,inout]*

Called from *test_lmloop()*, *lmloop_rdist()*

Call to *get_ximat_rdist()*, *get_openmp_blocks()*

subroutine updatexi_mod/**fill_ghosts_xi**(*xig*)

This subroutine is used to fill the ghosts zones that are located at nR=n_r_cmb-1 and nR=n_r_icb+1. This is used to properly set the Neuman boundary conditions. In case Dirichlet BCs are used, a simple first order extrapolation is employed. This is anyway only used for outputs (like Sherwood numbers).

Parameters **xig** (*lm_max*,3 - *nrstart* + *nrstop*) *[complex,inout]*

Called from *lmloop_rdist()*, *getstartfields()*, *start_from_another_scheme()*, *assemble_comp_rloc()*

Call to *get_openmp_blocks()*

subroutine updatexi_mod/**updatexi_fd**(*xi*, *dxidt*, *tscheme*)

This subroutine is called after the linear solves have been completed. This is then assembling the linear terms that will be used in the r.h.s. for the next iteration.

Parameters

- **xi** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]* :: Composition
- **dxidt** *[type_tarray,inout]*
- **tscheme** *[real]*

Called from *lmloop_rdist()*

Call to *get_comp_rhs_imp_ghost()*, *get_openmp_blocks()*

subroutine updatexi_mod/**finish_exp_comp**(*dvxirlm*, *dxi_exp_last*)

This subroutine completes the computation of the advection term which enters the composition equation by taking the radial derivative. This is the LM-distributed version.

Parameters

- **dvxirlm** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dxi_exp_last** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*

Called from *finish_explicit_assembly()*

Call to *get_openmp_blocks()*

subroutine updatexi_mod/**finish_exp_comp_rdist**(*dvxirlm*, *dxi_exp_last*)

This subroutine completes the computation of the advection term which enters the composition equation by taking the radial derivative. This is the R-distributed version.

Parameters

- **dvxirlm** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*
- **dxi_exp_last** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*

Called from *finish_explicit_assembly_rdist()*

Call to *get_dr_rloc()*, *get_openmp_blocks()*

subroutine updatexi_mod/**get_comp_rhs_imp**(*xi*, *dxi*, *dxidt*, *istage*, *l_calc_lin*[, *l_in_cheb_space*])

This subroutine computes the linear terms which enter the r.h.s. of the equation for composition. This is the LM-distributed version.

Parameters

- **xi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dxi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dxidt** *[type_tarray,inout]*
- **istage** *[integer,in]*
- **l_calc_lin** *[logical,in]*
- **l_in_cheb_space** *[logical,in,]*

Called from *readstartfields_old()*, *readstartfields()*, *readstartfields_mpi()*, *getstartfields()*, *start_from_another_scheme()*, *updatexi()*, *assemble_comp()*

Call to *get_openmp_blocks()*, *get_ddr()*

subroutine updatexi_mod/**get_comp_rhs_imp_ghost**(*xig*, *dxidt*, *istage*, *l_calc_lin*)

This subroutine computes the linear terms which enter the r.h.s. of the equation for composition. This is the R-distributed version.

Parameters

- **xig** (*lm_max*,3 - *nrstart* + *nrstop*) *[complex,inout]*
- **dxidt** *[type_tarray,inout]*
- **istage** *[integer,in]*
- **l_calc_lin** *[logical,in]*

Called from *getstartfields()*, *start_from_another_scheme()*, *updatexi_fd()*, *assemble_comp_rloc()*

Call to *get_openmp_blocks()*, *get_ddr_ghost()*

subroutine updatexi_mod/**assemble_comp**(*xi*, *dxi*, *dxidt*, *tscheme*)

This subroutine is used to assemble the chemical composition when an IMEX-RK with an assembly stage is employed. Non-Dirichlet boundary conditions are handled using Canuto (1986) approach. This is the LM distributed version.

Parameters

- **xi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dxi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dxidt** *[type_tarray,inout]*
- **tscheme** *[real]*

Called from *assemble_stage()*

Call to *get_comp_rhs_imp()*

subroutine updatexi_mod/**assemble_comp_rloc**(*xi*, *dxidt*, *tscheme*)

This subroutine is used when an IMEX Runge-Kutta time scheme with an assembly stage is used. This is used when R is distributed.

Parameters

- **xi** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*
- **dxidt** *[type_tarray,inout]*
- **tscheme** *[real]*

Called from *assemble_stage_rdist()*

Call to *get_openmp_blocks()*, *bulk_to_ghost()*, *exch_ghosts()*, *fill_ghosts_xi()*, *get_comp_rhs_imp_ghost()*

subroutine updatexi_mod/**get_xi0mat**(*tscheme*, *ximat*, *ximat_fac*)

Purpose of this subroutine is to contruct the time step matrix xiMat0

Parameters

- **tscheme** *[real]* :: time step
- **ximat** *[real]*
- **ximat_fac** (*n_r_max*) *[real,out]*

Called from *updatexi()*

Call to *abortrun()*

subroutine updatexi_mod/**get_ximat**(*tscheme*, *l*, *hdif*, *ximat*, *ximat_fac*)

Purpose of this subroutine is to contruct the time step matrices xiMat(i,j) for the equation for the chemical composition.

Parameters

- **tscheme** *[real]* :: time step
- **l** *[integer,in]*
- **hdif** *[real,in]*
- **ximat** *[real]* :: ‘)
- **ximat_fac** (*n_r_max*) *[real,out]*

Called from *updatexi()*

Call to *abortrun()*

subroutine updatexi_mod/**get_ximat_rdist**(*tscheme*, *hdif*, *ximat*)

Purpose of this subroutine is to contruct the time step matrices xiMat(i,j) for the equation for the chemical composition. This is used when parallel F.D. solvers are employed.

Parameters

- **tscheme** *[real]* :: time step
- **hdif** (1 + *l_max*) *[real,in]*

- **ximat** *[type_tri_par,inout]*

Called from *preparexi_fd()*

10.8.7 updateB.f90

Description

This module handles the time advance of the magnetic field potentials b and aj as well as the inner core counterparts b_ic and aj_ic. It contains the computation of the implicit terms and the linear solves.

Quick access

Variables *aj_ghost*, *b_ghost*, *bmat_fac*, *bmat_fd*, *dtp*, *dtt*, *jmat_fac*, *jmat_fd*, *lbmat*, *rhs2*, *work_ic_lmloc*, *worka*

Routines *assemble_mag()*, *assemble_mag_rloc()*, *fill_ghosts_b()*, *finalize_updateb()*, *finish_exp_mag()*, *finish_exp_mag_ic()*, *finish_exp_mag_rdist()*, *get_bmat()*, *get_bmat_rdist()*, *get_mag_ic_rhs_imp()*, *get_mag_rhs_imp()*, *get_mag_rhs_imp_ghost()*, *initialize_updateb()*, *prepareb_fd()*, *updateb()*, *updateb_fd()*

Needed modules

- omp_lib
- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *truncation* (*n_r_max()*, *n_r_tot()*, *n_r_ic_max()*, *n_cheb_ic_max()*, *n_r_ic_maxmag()*, *n_r_maxmag()*, *n_r_totmag()*, *lm_max()*, *l_maxmag()*, *lm_maxmag()*): This module defines the grid points and the truncation
- *radial_functions* (*chebt_ic()*, *or2()*, *r_cmb()*, *chebt_ic_even()*, *d2cheb_ic()*, *cheb_norm_ic()*, *dr_fac_ic()*, *lambda()*, *dllambda()*, *o_r_ic()*, *r()*, *or1()*, *cheb_ic()*, *dcheb_ic()*, rscheme_oc(), *dr_top_ic()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *radial_data* (*n_r_cmb()*, *n_r_icb()*, *nrstartmag()*, *nrstopmag()*): This module defines the MPI decomposition in the radial direction.
- *physical_parameters* (*n_r_lcr()*, *opm()*, *o_sr()*, *kbotb()*, *imagcon()*, *tmagcon()*, *sigma_ratio()*, *conductance_ma()*, *ktopb()*): Module containing the physical parameters
- *init_fields* (*bpeaktop()*, *bpeakbot()*): This module is used to construct the initial solution.
- *num_param* (*solve_counter()*, *dct_counter()*): Module containing numerical and control parameters
- *blocking* (*st_map()*, *lo_map()*, *st_sub_map()*, *lo_sub_map()*, *llmmag()*, *ulmmag()*): Module containing blocking information
- *horizontal_data* (*hdif_b()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

- *logic* (*l_cond_ic()*, *l_lcr()*, *l_rot_ic()*, *l_mag_nl()*, *l_b_nl_icb()*, *l_b_nl_cmb()*, *l_update_b()*, *l_rms()*, *l_finite_diff()*, *l_full_sphere()*, *l_mag_par_solve()*): Module containing the logicals that control the run
- *rms* (*dtbpollmr()*, *dtbpol2hint()*, *dtbtor2hint()*): This module contains the calculation of the RMS force balance and induction terms.
- *constants* (*pi()*, *zero()*, *one()*, *two()*, *three()*, *half()*): module containing constants and parameters used in the code.
- *special*: This module contains all variables for the case of an imposed IC dipole, an imposed external magnetic field and a special boundary forcing to excite inertial modes
- *parallel_mod* (rank(), *chunksize()*, *n_procs()*, *get_openmp_blocks()*): This module contains the blocking information
- *rms_helpers* (*hint2pollm()*, *hint2torlm()*): This module contains several useful subroutines required to compute RMS diagnostics
- *fields* (*work_lmloc()*): This module contains all the fields used in MagIC in the hybrid (LM,r) space as well as their radial derivatives. It defines both the LM-distributed arrays and the R-distributed arrays....
- *radial_der_even* (*get_ddr_even()*)
- *radial_der* (*get_dr()*, *get_ddr()*, *get_dr_rloc()*, *get_ddr_ghost()*, *exch_ghosts()*, *bulk_to_ghost()*): Radial derivatives functions
- *useful* (*abortrun()*): This module contains several useful routines.
- *time_schemes* (type_tscheme()): This module defines an abstract class type_tscheme which is employed for the time advance of the code.
- *time_array* (type_tarray()): This module defines two types that are defined to store the implicit/explicit terms at the different sub-stage/step.
- *dense_matrices*
- *band_matrices*
- bordered_matrices
- *real_matrices*
- *parallel_solvers* (type_tri_par()): This module contains the routines that are used to solve linear banded problems with R-distributed arrays.

Variables

- updateb_mod/**aj_ghost** (*,*) *[complex,public/allocatable]*
- updateb_mod/**b_ghost** (*,*) *[complex,public/allocatable]*
- updateb_mod/**bmat_fac** (*,*) *[real,private/allocatable]*
- updateb_mod/**bmat_fd** *[type_tri_par,public]*
- updateb_mod/**dtp** (*) *[complex,private/allocatable]*
- updateb_mod/**dtt** (*) *[complex,private/allocatable]*
- updateb_mod/**jmat_fac** (*,*) *[real,private/allocatable]*
- updateb_mod/**jmat_fd** *[type_tri_par,public]*
- updateb_mod/**lbmat** (*) *[logical,allocatable/public]*

- updateb_mod/**rhs1** (*,*,*) *[real,private/allocatable]*
- updateb_mod/**rhs2** (*,*,*) *[real,private/allocatable]*
- updateb_mod/**work_ic_lmloc** (*,*) *[complex,private/allocatable]*
- updateb_mod/**workA** (*,*) *[complex,private/allocatable]*
- updateb_mod/**workB** (*,*) *[complex,private/allocatable]*

Subroutines and functions

subroutine updateb_mod/**initialize_updateb**()

Purpose of this subroutine is to allocate the matrices needed to time advance the magnetic field. Depending on the radial scheme and the inner core conductivity, it can be full, bordered or band matrices.

Called from *initialize_lmloop()*

subroutine updateb_mod/**finalize_updateb**()

This subroutine deallocates the matrices involved in the time advance of the magnetic field.

Called from *finalize_lmloop()*

subroutine updateb_mod/**updateb**(*b*, *db*, *ddb*, *aj*, *dj*, *ddj*, *dbdt*, *djdt*, *b_ic*, *db_ic*, *ddb_ic*, *aj_ic*, *dj_ic*, *ddj_ic*, *dbdt_ic*, *djdt_ic*, *b_nl_cmb*, *aj_nl_cmb*, *aj_nl_icb*, *time*, *tscheme*, *lrmsnext*)

Calculates update of magnetic field potentials.

Parameters

- **b** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,inout]*
- **db** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,out]*
- **ddb** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,out]*
- **aj** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,inout]*
- **dj** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,out]*
- **ddj** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,out]*
- **dbdt** *[type_tarray,inout]*
- **djdt** *[type_tarray,inout]*
- **b_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,inout]*
- **db_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,out]*
- **ddb_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,out]*
- **aj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,inout]*
- **dj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,out]*
- **ddj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,out]*
- **dbdt_ic** *[type_tarray,inout]*
- **djdt_ic** *[type_tarray,inout]*
- **b_nl_cmb** (*) *[complex,in]* :: nonlinear BC for b at CMB
- **aj_nl_cmb** (*) *[complex,in]* :: nonlinear BC for aj at CMB
- **aj_nl_icb** (*) *[complex,in]* :: nonlinear BC for dr aj at ICB

- **time** *[real,in]*
- **tscheme** *[real]*
- **lrmsnext** *[logical,in]*

Called from *lmloop()*, *lmloop_rdist()*

Call to *abortrun()*, *get_bmat()*, *get_mag_rhs_imp()*, *get_mag_ic_rhs_imp()*

subroutine updateb_mod/**prepareb_fd**(*time*, *tscheme*, *dbdt*, *djdt*)

This subroutine assembles the r.h.s. when finite difference parallel solver is employed

Parameters

- **time** *[real,in]*
- **tscheme** *[real]*
- **dbdt** *[type_tarray,inout]*
- **djdt** *[type_tarray,inout]*

Called from *test_lmloop()*, *lmloop_rdist()*

Call to *get_bmat_rdist()*, *get_openmp_blocks()*, *abortrun()*

subroutine updateb_mod/**fill_ghosts_b**(*bg*, *ajg*)

This subroutine is used to fill the ghosts zones that are located at nR=n_r_cmb-1 and nR=n_r_icb+1. This is used to properly set the Neuman boundary conditions. In case Dirichlet BCs are used, a simple first order extrapolation is employed. This is anyway only used for outputs.

Parameters

- **bg** (*lm_max*,3 - *nrstartmag* + *nrstopmag*) *[complex,inout]*
- **ajg** (*lm_max*,3 - *nrstartmag* + *nrstopmag*) *[complex,inout]*

Called from *lmloop_rdist()*, *getstartfields()*, *start_from_another_scheme()*, *assemble_mag_rloc()*

Call to *get_openmp_blocks()*, *abortrun()*

subroutine updateb_mod/**updateb_fd**(*b*, *db*, *ddb*, *aj*, *dj*, *ddj*, *dbdt*, *djdt*, *tscheme*, *lrmsnext*)

This subroutine handles the IMEX postprocs once the solve has been completed

Parameters

- **b** (*lm_max*,1 - *nrstartmag* + *nrstopmag*) *[complex,inout]*
- **db** (*lm_max*,1 - *nrstartmag* + *nrstopmag*) *[complex,out]*
- **ddb** (*lm_max*,1 - *nrstartmag* + *nrstopmag*) *[complex,out]*
- **aj** (*lm_max*,1 - *nrstartmag* + *nrstopmag*) *[complex,inout]*
- **dj** (*lm_max*,1 - *nrstartmag* + *nrstopmag*) *[complex,out]*
- **ddj** (*lm_max*,1 - *nrstartmag* + *nrstopmag*) *[complex,out]*
- **dbdt** *[type_tarray,inout]*

- **djdt** *[type_tarray,inout]*
- **tscheme** *[real]*
- **lrmsnext** *[logical,in]*

Called from *lmloop_rdist()*

Call to *get_mag_rhs_imp_ghost()*, *get_openmp_blocks()*

subroutine updateb_mod/**finish_exp_mag_ic**(*b_ic*, *aj_ic*, *omega_ic*, *db_exp_last*, *dj_exp_last*)

This subroutine computes the nonlinear term at the inner core boundary when there is a conducting inner core and stress-free boundary conditions.

Parameters

- **b_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_max*) *[complex,in]*
- **aj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_max*) *[complex,in]*
- **omega_ic** *[real,in]*
- **db_exp_last** (1 - *llmmag* + *ulmmag*,*n_r_ic_max*) *[complex,inout]*
- **dj_exp_last** (1 - *llmmag* + *ulmmag*,*n_r_ic_max*) *[complex,inout]*

Called from *finish_explicit_assembly()*, *finish_explicit_assembly_rdist()*

subroutine updateb_mod/**finish_exp_mag**(*dvxbhlm*, *dj_exp_last*)

This subroutine finishes the computation of the nonlinear induction term by taking the missing radial derivative (LM-distributed version).

Parameters

- **dvxbhlm** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,inout]*
- **dj_exp_last** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,inout]*

Called from *finish_explicit_assembly()*

Call to *get_openmp_blocks()*

subroutine updateb_mod/**finish_exp_mag_rdist**(*dvxbhlm*, *dj_exp_last*)

This subroutine finishes the computation of the nonlinear induction term by taking the missing radial derivative (R-distributed version).

Parameters

- **dvxbhlm** (*lm_max*,1 - *nrstartmag* + *nrstopmag*) *[complex,inout]*
- **dj_exp_last** (*lm_max*,1 - *nrstartmag* + *nrstopmag*) *[complex,inout]*

Called from *finish_explicit_assembly_rdist()*

Call to *get_dr_rloc()*, *get_openmp_blocks()*

subroutine updateb_mod/**get_mag_ic_rhs_imp**(*b_ic*, *db_ic*, *ddb_ic*, *aj_ic*, *dj_ic*, *ddj_ic*, *dbdt_ic*, *djdt_ic*, *istage*, *l_calc_lin*[, *l_in_cheb_space*])

This subroutine computes the linear terms that enter the r.h.s. of the inner core equations.

Parameters

- **b_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_max*) *[complex,inout]*
- **db_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_max*) *[complex,out]*
- **ddb_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_max*) *[complex,out]*
- **aj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_max*) *[complex,inout]*
- **dj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_max*) *[complex,out]*
- **ddj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_max*) *[complex,out]*
- **dbdt_ic** *[type_tarray,inout]*
- **djdt_ic** *[type_tarray,inout]*
- **istage** *[integer,in]*
- **l_calc_lin** *[logical,in]*
- **l_in_cheb_space** *[logical,in,]*

Called from *readstartfields_old()*, *readstartfields()*, *readstartfields_mpi()*, *getstartfields()*, *start_from_another_scheme()*, *updateb()*, *assemble_mag()*

Call to *get_openmp_blocks()*, *get_ddr_even()*

subroutine updateb_mod/**assemble_mag**(*b*, *db*, *ddb*, *aj*, *dj*, *ddj*, *b_ic*, *db_ic*, *ddb_ic*, *aj_ic*, *dj_ic*, *ddj_ic*, *dbdt*, *djdt*, *dbdt_ic*, *djdt_ic*, *lrmsnext*, *tscheme*)

This subroutine is used when an IMEX Runge Kutta with an assembly stage is employed. This is the LM-distributed version

Parameters

- **b** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,inout]*
- **db** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,out]*
- **ddb** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,out]*
- **aj** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,inout]*
- **dj** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,out]*
- **ddj** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,out]*
- **b_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_max*) *[complex,inout]*
- **db_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_max*) *[complex,out]*
- **ddb_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_max*) *[complex,out]*
- **aj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_max*) *[complex,inout]*
- **dj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_max*) *[complex,out]*
- **ddj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_max*) *[complex,out]*
- **dbdt** *[type_tarray,inout]*
- **djdt** *[type_tarray,inout]*
- **dbdt_ic** *[type_tarray,inout]*
- **djdt_ic** *[type_tarray,inout]*

- **lrmsnext** *[logical,in]*
- **tscheme** *[real]*

Called from assemble_stage(), assemble_stage_rdist()

Call to abortrun(), get_mag_rhs_imp(), get_mag_ic_rhs_imp()

subroutine updateb_mod/**assemble_mag_rloc**(*b*, *db*, *ddb*, *aj*, *dj*, *ddj*, *dbdt*, *djdt*, *lrmsnext*, *tscheme*)

This subroutine is used when an IMEX Runge Kutta with an assembly stage is employed. This is the R-distributed version.

Parameters

- **b** (*lm_maxmag*,1 - *nrstartmag* + *nrstopmag*) *[complex,inout]*
- **db** (*lm_maxmag*,1 - *nrstartmag* + *nrstopmag*) *[complex,out]*
- **ddb** (*lm_maxmag*,1 - *nrstartmag* + *nrstopmag*) *[complex,out]*
- **aj** (*lm_maxmag*,1 - *nrstartmag* + *nrstopmag*) *[complex,inout]*
- **dj** (*lm_maxmag*,1 - *nrstartmag* + *nrstopmag*) *[complex,out]*
- **ddj** (*lm_maxmag*,1 - *nrstartmag* + *nrstopmag*) *[complex,out]*
- **dbdt** *[type_tarray,inout]*
- **djdt** *[type_tarray,inout]*
- **lrmsnext** *[logical,in]*
- **tscheme** *[real]*

Called from assemble_stage_rdist()

Call to abortrun(), get_openmp_blocks(), bulk_to_ghost(), exch_ghosts(), fill_ghosts_b(), get_mag_rhs_imp_ghost()

subroutine updateb_mod/**get_mag_rhs_imp**(*b*, *db*, *ddb*, *aj*, *dj*, *ddj*, *dbdt*, *djdt*, *tscheme*, *istage*, *l_calc_lin*, *lrmsnext*[, *l_in_cheb_space*])

This subroutine handles the computation of the linear terms which enter the r.h.s. of the induction equation (LM-distributed version).

Parameters

- **b** (1 - *llmmag* + *ulmmag*,*n_r_max*) *[complex,inout]*
- **db** (1 - *llmmag* + *ulmmag*,*n_r_max*) *[complex,out]*
- **ddb** (1 - *llmmag* + *ulmmag*,*n_r_max*) *[complex,out]*
- **aj** (1 - *llmmag* + *ulmmag*,*n_r_max*) *[complex,inout]*
- **dj** (1 - *llmmag* + *ulmmag*,*n_r_max*) *[complex,out]*
- **ddj** (1 - *llmmag* + *ulmmag*,*n_r_max*) *[complex,out]*
- **dbdt** *[type_tarray,inout]*
- **djdt** *[type_tarray,inout]*
- **tscheme** *[real]*

- **istage** *[integer,in]*
- **l_calc_lin** *[logical,in]*
- **lrmsnext** *[logical,in]*
- **l_in_cheb_space** *[logical,in,]*

Called from *readstartfields_old()*, *readstartfields()*, *readstartfields_mpi()*, *getstartfields()*, *start_from_another_scheme()*, *updateb()*, *assemble_mag()*

Call to *get_openmp_blocks()*, *get_ddr()*, *hint2pollm()*, *hint2torlm()*

subroutine updateb_mod/**get_mag_rhs_imp_ghost**(*bg*, *db*, *ddb*, *ajg*, *dj*, *ddj*, *dbdt*, *djdt*, *tscheme*, *istage*, *l_calc_lin*, *lrmsnext*)

This subroutine handles the computation of the linear terms which enter the r.h.s. of the induction equation (R-distributed version).

Parameters

- **bg** (*lm_max*,3 - *nrstartmag* + *nrstopmag*) *[complex,inout]*
- **db** (*lm_max*,1 - *nrstartmag* + *nrstopmag*) *[complex,out]*
- **ddb** (*lm_max*,1 - *nrstartmag* + *nrstopmag*) *[complex,out]*
- **ajg** (*lm_max*,3 - *nrstartmag* + *nrstopmag*) *[complex,inout]*
- **dj** (*lm_max*,1 - *nrstartmag* + *nrstopmag*) *[complex,out]*
- **ddj** (*lm_max*,1 - *nrstartmag* + *nrstopmag*) *[complex,out]*
- **dbdt** *[type_tarray,inout]*
- **djdt** *[type_tarray,inout]*
- **tscheme** *[real]*
- **istage** *[integer,in]*
- **l_calc_lin** *[logical,in]*
- **lrmsnext** *[logical,in]*

Called from *getstartfields()*, *start_from_another_scheme()*, *updateb_fd()*, *assemble_mag_rloc()*

Call to *get_openmp_blocks()*, *get_ddr_ghost()*, *hint2pollm()*, *hint2torlm()*

subroutine updateb_mod/**get_bmat**(*tscheme*, *l*, *hdif*, *bmat*, *bmat_fac*, *jmat*, *jmat_fac*)

Purpose of this subroutine is to contruct the time step matrices `bmat(i,j)` and `ajmat` for the dynamo equations.

Parameters

- **tscheme** *[real]* :: time step
- **l** *[integer,in]*
- **hdif** *[real,in]*
- **bmat** *[real]*
- **bmat_fac** (*n_r_totmag*) *[real,out]*

- **jmat** *[integer]* :: ‘)
- **jmat_fac** (*n_r_totmag*) *[real,out]*

Called from *updateb()*

Call to *abortrun()*

subroutine updateb_mod/**get_bmat_rdist**(*tscheme*, *hdif*, *bmat*, *jmat*)

Purpose of this subroutine is to contruct the time step matrices `bmat(i,j)` and `ajmat` for the dynamo equations when the parallel F.D. solver is used

Parameters

- **tscheme** *[real]* :: time step
- **hdif** (1 + *l_maxmag*) *[real,in]*
- **bmat** *[type_tri_par,inout]*
- **jmat** *[type_tri_par,inout]*

Called from *prepareb_fd()*

Call to *abortrun()*

10.8.8 updatePHI.f90

Description

This module handles the time advance of the phase field phi. It contains the computation of the implicit terms and the linear solves.

Quick access

Variables *lphimat*, *phi0mat_fac*, *phi_ghost*, *phimat_fac*, *phimat_fd*

Routines *assemble_phase()*, *assemble_phase_rloc()*, *fill_ghosts_phi()*, *finalize_updatephi()*, *get_phase_rhs_imp()*, *get_phase_rhs_imp_ghost()*, *get_phi0mat()*, *get_phimat()*, *get_phimat_rdist()*, *initialize_updatephi()*, *preparephase_fd()*, *updatephase_fd()*, *updatephi()*

Needed modules

- `omp_lib`
- *precision_mod*: This module controls the precision used in MagIC
- *truncation* (*n_r_max()*, *lm_max()*, *l_max()*): This module defines the grid points and the truncation
- *radial_data* (*n_r_icb()*, *n_r_cmb()*, *nrstart()*, *nrstop()*): This module defines the MPI decomposition in the radial direction.
- *radial_functions* (*or2()*, `rscheme_oc()`, *r()*, *or1()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *num_param* (*dct_counter()*, *solve_counter()*): Module containing numerical and control parameters

- *physical_parameters* (*pr()*, *phasediffffac()*, *stef()*, *ktopphi()*, *kbotphi()*): Module containing the physical parameters
- *init_fields* (*phi_top()*, *phi_bot()*): This module is used to construct the initial solution.
- *blocking* (*lo_map()*, *lo_sub_map()*, *llm()*, *ulm()*, *st_map()*): Module containing blocking information
- *logic* (*l_finite_diff()*, *l_full_sphere()*, *l_parallel_solve()*): Module containing the logicals that control the run
- *parallel_mod* (rank(), *chunksize()*, *n_procs()*, *get_openmp_blocks()*): This module contains the blocking information
- *radial_der* (*get_ddr()*, *get_ddr_ghost()*, *exch_ghosts()*, *bulk_to_ghost()*): Radial derivatives functions
- *constants* (*zero()*, *one()*, *two()*): module containing constants and parameters used in the code.
- *fields* (*work_lmloc()*): This module contains all the fields used in MagIC in the hybrid (LM,r) space as well as their radial derivatives. It defines both the LM-distributed arrays and the R-distributed arrays....
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *useful* (*abortrun()*): This module contains several useful routines.
- *time_schemes* (type_tscheme()): This module defines an abstract class type_tscheme which is employed for the time advance of the code.
- *time_array* (type_tarray()): This module defines two types that are defined to store the implicit/explicit terms at the different sub-stage/step.
- *dense_matrices*
- *real_matrices*
- *band_matrices*
- *parallel_solvers* (type_tri_par()): This module contains the routines that are used to solve linear banded problems with R-distributed arrays.

Variables

- updatephi_mod/**lphimat** (*) *[logical,allocatable/public]*
- updatephi_mod/**maxthreads** *[integer,private]*
- updatephi_mod/**phi0mat_fac** (*) *[real,private/allocatable]*
- updatephi_mod/**phi_ghost** (*,*) *[complex,allocatable/public]*
- updatephi_mod/**phimat_fac** (*,*) *[real,private/allocatable]*
- updatephi_mod/**phimat_fd** *[type_tri_par,public]*
- updatephi_mod/**rhs1** (*,*,*) *[real,private/allocatable]*

Subroutines and functions

subroutine updatephi_mod/**initialize_updatephi**()

Called from *initialize_lmloop()*

subroutine updatephi_mod/**finalize_updatephi**()
This subroutine deallocates the matrices involved in the time-advance of phi.

Called from *finalize_lmloop()*

subroutine updatephi_mod/**updatephi**(*phi*, *dphidt*, *tscheme*)

Updates the phase field

Parameters

- **phi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]* :: Chemical composition
- **dphidt** *[type_tarray,inout]*
- **tscheme** *[real]*

Called from *lmloop()*

Call to *get_phi0mat()*, *get_phimat()*, *get_phase_rhs_imp()*

subroutine updatephi_mod/**preparephase_fd**(*tscheme*, *dphidt*)

This subroutine is used to assemble the r.h.s. of the phase field equation when parallel F.D solvers are used. Boundary values are set here.

Parameters

- **tscheme** *[real]*
- **dphidt** *[type_tarray,inout]*

Called from *lmloop_rdist()*

Call to *get_phimat_rdist()*, *get_openmp_blocks()*

subroutine updatephi_mod/**fill_ghosts_phi**(*phig*)

This subroutine is used to fill the ghosts zones that are located at nR=n_r_cmb-1 and nR=n_r_icb+1. This is used to properly set the Neuman boundary conditions. In case Dirichlet BCs are used, a simple first order extrapolation is employed. This is anyway only used for outputs (like Sherwood numbers).

Parameters **phig** (*lm_max*,3 - *nrstart* + *nrstop*) *[complex,inout]*

Called from *lmloop_rdist()*, *getstartfields()*, *start_from_another_scheme()*, *assemble_phase_rloc()*

Call to *get_openmp_blocks()*

subroutine updatephi_mod/**updatephase_fd**(*phi*, *dphidt*, *tscheme*)

This subroutine is called after the linear solves have been completed. This is then assembling the linear terms that will be used in the r.h.s. for the next iteration.

Parameters

- **phi** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]* :: Phase field
- **dphidt** *[type_tarray,inout]*
- **tscheme** *[real]*

Called from *lmloop_rdist()*

Call to *get_phase_rhs_imp_ghost()*, *get_openmp_blocks()*

subroutine updatephi_mod/**get_phase_rhs_imp**(*phi*, *dphidt*, *istage*, *l_calc_lin*[, *l_in_cheb_space*])

This subroutine computes the linear terms which enter the r.h.s. of the equation for phase field. This is the LM-distributed version.

Parameters

- **phi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dphidt** *[type_tarray,inout]*
- **istage** *[integer,in]*
- **l_calc_lin** *[logical,in]*
- **l_in_cheb_space** *[logical,in,]*

Called from *getstartfields()*, *start_from_another_scheme()*, *updatephi()*, *assemble_phase()*

Call to *get_openmp_blocks()*, *get_ddr()*

subroutine updatephi_mod/**get_phase_rhs_imp_ghost**(*phig*, *dphidt*, *istage*, *l_calc_lin*)

This subroutine computes the linear terms which enter the r.h.s. of the equation for phase field. This is the R-distributed version.

Parameters

- **phig** (*lm_max*,3 - *nrstart* + *nrstop*) *[complex,inout]*
- **dphidt** *[type_tarray,inout]*
- **istage** *[integer,in]*
- **l_calc_lin** *[logical,in]*

Called from *getstartfields()*, *start_from_another_scheme()*, *updatephase_fd()*, *assemble_phase_rloc()*

Call to *get_openmp_blocks()*, *get_ddr_ghost()*

subroutine updatephi_mod/**assemble_phase**(*phi*, *dphidt*, *tscheme*)

This subroutine is used to assemble the phase field when an IMEX-RK with an assembly stage is employed.

Parameters

- **phi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **dphidt** *[type_tarray,inout]*

- **tscheme** *[real]*

Called from *assemble_stage()*

Call to *get_phase_rhs_imp()*

subroutine updatephi_mod/**assemble_phase_rloc**(*phi*, *dphidt*, *tscheme*)

This subroutine is used when an IMEX Runge-Kutta time scheme with an assembly stage is used. This is used when R is distributed.

Parameters

- **phi** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]*
- **dphidt** *[type_tarray,inout]*
- **tscheme** *[real]*

Called from *assemble_stage_rdist()*

Call to *get_openmp_blocks()*, *bulk_to_ghost()*, *exch_ghosts()*, *fill_ghosts_phi()*, *get_phase_rhs_imp_ghost()*

subroutine updatephi_mod/**get_phi0mat**(*tscheme*, *phimat*, *phimat_fac*)

Purpose of this subroutine is to contruct the time step matrix phiMat0

Parameters

- **tscheme** *[real]* :: time step
- **phimat** *[real]*
- **phimat_fac** (*n_r_max*) *[real,out]*

Called from *updatephi()*

Call to *abortrun()*

subroutine updatephi_mod/**get_phimat**(*tscheme*, *l*, *phimat*, *phimat_fac*)

Purpose of this subroutine is to contruct the time step matrices phiMat(i,j) for the equation for phase field.

Parameters

- **tscheme** *[real]* :: time step
- **l** *[integer,in]*
- **phimat** *[real]* :: ‘)
- **phimat_fac** (*n_r_max*) *[real,out]*

Called from *updatephi()*

Call to *abortrun()*

subroutine updatephi_mod/**get_phimat_rdist**(*tscheme*, *phimat*)

Purpose of this subroutine is to contruct the time step matrices phiMat(i,j) for the equation for the phase field. This is used when parallel F.D. solvers are employed.

Parameters

- **tscheme** *[real]* :: time step
- **phimat** *[type_tri_par,inout]*

Called from *preparephase_fd()*

10.9 Non-linear part of the time stepping (radial loop)

10.9.1 radialLoop.f90

Quick access

Routines *finalize_radialloop()*, *initialize_radialloop()*, *radialloopg()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc* (*memwrite()*, *bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *truncation* (*lm_max()*, *lm_maxmag()*, *l_max()*, *l_maxmag()*, *lmp_max()*): This module defines the grid points and the truncation
- *radial_data* (*nrstart()*, *nrstop()*, *nrstartmag()*, *nrstopmag()*): This module defines the MPI decomposition in the radial direction.
- *time_schemes* (`type_tscheme()`): This module defines an abstract class type_tscheme which is employed for the time advance of the code.
- *riteration* (`riter_t()`): This module is used to define an abstract class for the radial loop
- *riter_mod* (`riter_single_t()`): This module actually handles the loop over the radial levels. It contains the spherical harmonic transforms and the operations on the arrays in physical space....

Variables

Subroutines and functions

`subroutine radialloop/initialize_radialloop()`

Called from *magic*

Call to *memwrite()*

`subroutine radialloop/finalize_radialloop()`

Called from *magic*

subroutine radialloop/**radialloopg**(*l_graph*, *l_frame*, *time*, *timestage*, *tscheme*, *dtlast*, *ltocalc*, *ltonext*, *ltonext2*, *lhelcalc*, *lpowercalc*, *lrmscalc*, *lpresscalc*, *lpressnext*, *lviscbccalc*, *lfluxprofcalc*, *lperpparcalc*, *lgeoscalc*, *l_probe_out*, *dsdt*, *dwdt*, *dzdt*, *dpdt*, *dxidt*, *dphidt*, *dbdt*, *djdt*, *dvxvhlm*, *dvxbhlm*, *dvsrlm*, *dvxirlm*, *lorentz_torque_ic*, *lorentz_torque_ma*, *br_vt_lm_cmb*, *br_vp_lm_cmb*, *br_vt_lm_icb*, *br_vp_lm_icb*, *helas*, *hel2as*, *helnaas*, *helna2as*, *heleaas*, *viscas*, *uhas*, *duhas*, *gradsas*, *fconvas*, *fkinas*, *fviscas*, *fpoynas*, *fresas*, *eperpas*, *eparas*, *eperpaxias*, *eparaxias*, *ekins*, *ekinl*, *vols*, *dtrkc*, *dthkc*)

This subroutine performs the actual time-stepping.

Parameters

- **l_graph** *[logical,in]*
- **l_frame** *[logical,in]*
- **time** *[real,in]*
- **timestage** *[real,in]*
- **tscheme** *[real]*
- **dtlast** *[real,in]*
- **ltocalc** *[logical,in]*
- **ltonext** *[logical,in]*
- **ltonext2** *[logical,in]*
- **lhelcalc** *[logical,in]*
- **lpowercalc** *[logical,in]*
- **lrmscalc** *[logical,in]*
- **lpresscalc** *[logical,in]*
- **lpressnext** *[logical,in]*
- **lviscbccalc** *[logical,in]*
- **lfluxprofcalc** *[logical,in]*
- **lperpparcalc** *[logical,in]*
- **lgeoscalc** *[logical,in]*
- **l_probe_out** *[logical,in]*
- **dsdt** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **dwdt** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **dzdt** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **dpdt** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **dxidt** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **dphidt** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **dbdt** (*lm_maxmag*,1 - *nrstartmag* + *nrstopmag*) *[complex,out]*
- **djdt** (*lm_maxmag*,1 - *nrstartmag* + *nrstopmag*) *[complex,out]*

- **dvxvhlm** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **dvxbhlm** (*lm_maxmag*,1 - *nrstartmag* + *nrstopmag*) *[complex,out]*
- **dvsrlm** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **dvxirlm** (*lm_max*,1 - *nrstart* + *nrstop*) *[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
- **helas** (2,1 - *nrstart* + *nrstop*) *[real,out]*
- **hel2as** (2,1 - *nrstart* + *nrstop*) *[real,out]*
- **helnaas** (2,1 - *nrstart* + *nrstop*) *[real,out]*
- **helna2as** (2,1 - *nrstart* + *nrstop*) *[real,out]*
- **heleaas** (1 - *nrstart* + *nrstop*) *[real,out]*
- **viscas** (1 - *nrstart* + *nrstop*) *[real,out]*
- **uhas** (1 - *nrstart* + *nrstop*) *[real,out]*
- **duhas** (1 - *nrstart* + *nrstop*) *[real,out]*
- **gradsas** (1 - *nrstart* + *nrstop*) *[real,out]*
- **fconvas** (1 - *nrstart* + *nrstop*) *[real,out]*
- **fkinas** (1 - *nrstart* + *nrstop*) *[real,out]*
- **fviscas** (1 - *nrstart* + *nrstop*) *[real,out]*
- **fpoynas** (1 - *nrstartmag* + *nrstopmag*) *[real,out]*
- **fresas** (1 - *nrstartmag* + *nrstopmag*) *[real,out]*
- **eperpas** (1 - *nrstart* + *nrstop*) *[real,out]*
- **eparas** (1 - *nrstart* + *nrstop*) *[real,out]*
- **eperpaxias** (1 - *nrstart* + *nrstop*) *[real,out]*
- **eparaxias** (1 - *nrstart* + *nrstop*) *[real,out]*
- **ekins** (1 - *nrstart* + *nrstop*) *[real,out]*
- **ekinl** (1 - *nrstart* + *nrstop*) *[real,out]*
- **vols** (1 - *nrstart* + *nrstop*) *[real,out]*
- **dtrkc** (1 - *nrstart* + *nrstop*) *[real,out]*
- **dthkc** (1 - *nrstart* + *nrstop*) *[real,out]*

Called from `step_time()`

10.9.2 rIteration.f90

Description

This module is used to define an abstract class for the radial loop

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *radial_data* (*nrstart()*, *nrstop()*, *nrstartmag()*, *nrstopmag()*): This module defines the MPI decomposition in the radial direction.
- *truncation* (*lm_max()*, *lm_maxmag()*): This module defines the grid points and the truncation
- *time_schemes* (type_tscheme()): This module defines an abstract class type_tscheme which is employed for the time advance of the code.

Types

- **type** riteration/**unknown_type**

10.9.3 rIter.f90

Description

This module actually handles the loop over the radial levels. It contains the spherical harmonic transforms and the operations on the arrays in physical space.

Quick access

Routines *radialloop()*, *transform_to_grid_space()*, *transform_to_lm_space()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *num_param* (*phy2lm_counter()*, *lm2phy_counter()*, *nl_counter()*, *td_counter()*): Module containing numerical and control parameters
- *parallel_mod*: This module contains the blocking information
- *truncation* (*lmp_max()*, *n_phi_max()*, *lm_max()*, *lm_maxmag()*): This module defines the grid points and the truncation
- *logic* (*l_mag()*, *l_conv()*, *l_mag_kin()*, *l_heat()*, *l_ht()*, *l_anel()*, *l_mag_lf()*, *l_conv_nl()*, *l_mag_nl()*, *l_b_nl_cmb()*, *l_b_nl_icb()*, *l_rot_ic()*, *l_cond_ic()*, *l_rot_ma()*, *l_cond_ma()*, *l_dtb()*, *l_store_frame()*, *l_movie_oc()*, *l_to()*, *l_chemical_conv()*, *l_probe()*, *l_full_sphere()*, *l_precession()*, *l_centrifuge()*, *l_adv_curl()*, *l_double_curl()*, *l_parallel_solve()*, *l_single_matrix()*, *l_temperature_diff()*, *l_rms()*, *l_phase_field()*): Module containing the logicals that control the run
- *radial_data* (*n_r_cmb()*, *n_r_icb()*, *nrstart()*, *nrstop()*, *nrstartmag()*, *nrstopmag()*): This module defines the MPI decomposition in the radial direction.

- *radial_functions* (*or2()*, *orho1()*, *l_r()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *constants* (*zero()*): module containing constants and parameters used in the code.
- *nonlinear_lm_mod* (nonlinear_lm_t()): This module is used to finish the assembly of the nonlinear terms in (ℓ, m) space. Derivatives along θ and ϕ are handled using recurrence relations....
- grid_space_arrays_mod (grid_space_arrays_t()): This module is used to compute the nonlinear products in physical space (θ, ϕ).
- *to_arrays_mod* (to_arrays_t())
- *dtb_arrays_mod* (dtb_arrays_t())
- *torsional_oscillations* (*prep_to_axi()*, *getto()*, *gettonext()*, *gettofinish()*): This module contains information for TO calculation and output
- *graphout_mod* (*graphout_mpi()*, *graphout_mpi_header()*): This module contains the subroutines that store the 3-D graphic files.
- *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....
- *out_movie* (*store_movie_frame()*)
- *outrot* (*get_lorentz_torque()*): This module handles the writing of several diagnostic files related to the rotation: angular momentum (AM.TAG), drift (drift.TAG), inner core and mantle rotations....
- *courant_mod* (*courant()*): This module handles the computation of Courant factors on grid space. It then checks whether the timestep size needs to be modified.
- *nonlinear_bcs* (*get_br_v_bcs()*, *v_rigid_boundary()*)
- *nl_special_calc*: This module allows to calculcate several diagnostics that need to be computed in the physical space (non-linear quantities)
- *geos* (*calcgeos()*): This module is used to compute z-integrated diagnostics such as the degree of geostrophy or the separation of energies between inside and outside the tangent cylinder. This makes use of a local Simpson's method. This also
- *sht*
- *fields* (*s_rloc()*, *ds_rloc()*, *z_rloc()*, *dz_rloc()*, *p_rloc()*, *b_rloc()*, *db_rloc()*, *ddb_rloc()*, *aj_rloc()*, *dj_rloc()*, *w_rloc()*, *dw_rloc()*, *ddw_rloc()*, *xi_rloc()*, *omega_ic()*, *omega_ma()*, *dp_rloc()*, *phi_rloc()*): This module contains all the fields used in MagIC in the hybrid (LM,r) space as well as their radial derivatives. It defines both the LM-distributed arrays and the R-distributed arrays....
- *time_schemes* (type_tscheme()): This module defines an abstract class type_tscheme which is employed for the time advance of the code.
- *physical_parameters* (*ktops()*, *kbots()*, *n_r_lcr()*, *ktopv()*, *kbotv()*): Module containing the physical parameters
- *riteration* (riter_t()): This module is used to define an abstract class for the radial loop
- *rms* (*get_nl_rms()*, *transform_to_lm_rms()*, *compute_lm_forces()*, *transform_to_grid_rms()*): This module contains the calculation of the RMS force balance and induction terms.
- *probe_mod*

Types

- **type** riter_mod/**unknown_type**

  Type fields

  - **% dtb_arrays** *[dtb_arrays_t]*
  - **% gsa** *[grid_space_arrays_t]*
  - **% nl_lm** *[nonlinear_lm_t]*
  - **% to_arrays** *[to_arrays_t]*

Subroutines and functions

subroutine riter_mod/**initialize**(*this*)

Parameters **this** *[real]*

subroutine riter_mod/**finalize**(*this*)

Parameters **this** *[real]*

subroutine riter_mod/**radialloop**(*this*, *l_graph*, *l_frame*, *time*, *timestage*, *tscheme*, *dtlast*, *ltocalc*, *ltonext*, *ltonext2*, *lhelcalc*, *lpowercalc*, *lrmscalc*, *lpresscalc*, *lpressnext*, *lviscbccalc*, *lfluxprofcalc*, *lperpparcalc*, *lgeoscalc*, *l_probe_out*, *dsdt*, *dwdt*, *dzdt*, *dpdt*, *dxidt*, *dphidt*, *dbdt*, *djdt*, *dvxvhlm*, *dvxbhlm*, *dvsrlm*, *dvxirlm*, *lorentz_torque_ic*, *lorentz_torque_ma*, *br_vt_lm_cmb*, *br_vp_lm_cmb*, *br_vt_lm_icb*, *br_vp_lm_icb*, *helas*, *hel2as*, *helnaas*, *helna2as*, *heleaas*, *viscas*, *uhas*, *duhas*, *gradsas*, *fconvas*, *fkinas*, *fviscas*, *fpoynas*, *fresas*, *eperpas*, *eparas*, *eperpaxias*, *eparaxias*, *ekins*, *ekinl*, *vols*, *dtrkc*, *dthkc*)

This subroutine handles the main loop over the radial levels. It calls the SH transforms, computes the nonlinear terms on the grid and bring back the quantities in spectral space. This is the most time consuming part of MagIC.

Parameters

- **this** *[real]*
- **l_graph** *[logical,in]*
- **l_frame** *[logical,in]*
- **time** *[real,in]*
- **timestage** *[real,in]*
- **tscheme** *[real]*
- **dtlast** *[real,in]*
- **ltocalc** *[logical,in]*
- **ltonext** *[logical,in]*
- **ltonext2** *[logical,in]*
- **lhelcalc** *[logical,in]*

- **lpowercalc** *[logical,in]*
- **lrmscalc** *[logical,in]*
- **lpresscalc** *[logical,in]*
- **lpressnext** *[logical,in]*
- **lviscbccalc** *[logical,in]*
- **lfluxprofcalc** *[logical,in]*
- **lperpparcalc** *[logical,in]*
- **lgeoscalc** *[logical,in]*
- **l_probe_out** *[logical,in]*
- **dsdt** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **dwdt** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **dzdt** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **dpdt** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **dxidt** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **dphidt** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **dbdt** (*lm_maxmag*,1 - *nrstartmag* + *nrstopmag*) *[complex,out]*
- **djdt** (*lm_maxmag*,1 - *nrstartmag* + *nrstopmag*) *[complex,out]*
- **dvxvhlm** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **dvxbhlm** (*lm_maxmag*,1 - *nrstartmag* + *nrstopmag*) *[complex,out]*
- **dvsrlm** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **dvxirlm** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **lorentz_torque_ic** *[real,out]*
- **lorentz_torque_ma** *[real,out]*
- **br_vt_lm_cmb** (*) *[complex,out]* :: product br*vt at CMB
- **br_vp_lm_cmb** (*) *[complex,out]* :: product br*vp at CMB
- **br_vt_lm_icb** (*) *[complex,out]* :: product br*vt at ICB
- **br_vp_lm_icb** (*) *[complex,out]* :: product br*vp at ICB
- **helas** (2,1 - *nrstart* + *nrstop*) *[real,inout]*
- **hel2as** (2,1 - *nrstart* + *nrstop*) *[real,inout]*
- **helnaas** (2,1 - *nrstart* + *nrstop*) *[real,inout]*
- **helna2as** (2,1 - *nrstart* + *nrstop*) *[real,inout]*
- **heleaas** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **viscas** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **uhas** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **duhas** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **gradsas** (1 - *nrstart* + *nrstop*) *[real,inout]*

- **fconvas** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **fkinas** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **fviscas** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **fpoynas** (1 - *nrstartmag* + *nrstopmag*) *[real,inout]*
- **fresas** (1 - *nrstartmag* + *nrstopmag*) *[real,inout]*
- **eperpas** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **eparas** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **eperpaxias** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **eparaxias** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **ekins** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **ekinl** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **vols** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **dtrkc** (1 - *nrstart* + *nrstop*) *[real,out]*
- **dthkc** (1 - *nrstart* + *nrstop*) *[real,out]*

Call to *graphout_mpi_header()*, *prep_to_axi()*, *get_nl_rms()*, *get_br_v_bcs()*, *get_lorentz_torque()*, *courant()*, *graphout_mpi()*, *probe_out()*, *get_helicity()*, *get_visc_heat()*, *get_nlblayers()*, *get_fluxes()*, *get_ekin_solid_liquid()*, *get_perppar()*, *calcgeos()*, *store_movie_frame()*, *get_dtblm()*, *gettonext()*, *getto()*, *compute_lm_forces()*, *gettofinish()*, *get_dh_dtblm()*

subroutine riter_mod/**transform_to_grid_space**(*this*, *nr*, *nbc*, *lviscbccalc*, *lrmscalc*, *lpresscalc*, *ltocalc*, *lpowercalc*, *lfluxprofcalc*, *lperpparcalc*, *lhelcalc*, *lgeoscalc*, *l_frame*, *lderiv*)

This subroutine actually handles the spherical harmonic transforms from (ell,m) space to (theta,phi) space.

Parameters

- **this** *[real]*
- **nr** *[integer,in]*
- **nbc** *[integer,in]*
- **lviscbccalc** *[logical,in]*
- **lrmscalc** *[logical,in]*
- **lpresscalc** *[logical,in]*
- **ltocalc** *[logical,in]*
- **lpowercalc** *[logical,in]*
- **lfluxprofcalc** *[logical,in]*
- **lperpparcalc** *[logical,in]*
- **lhelcalc** *[logical,in]*
- **lgeoscalc** *[logical,in]*

- **l_frame** *[logical,in]*
- **lderiv** *[logical,in]*

Call to *scal_to_spat()*, *scal_to_grad_spat()*, *transform_to_grid_rms()*, *torpol_to_spat()*, *torpol_to_curl_spat()*, *pol_to_grad_spat()*, *torpol_to_dphspat()*, *pol_to_curlr_spat()*, *v_rigid_boundary()*

subroutine riter_mod/**transform_to_lm_space**(*this*, *nr*, *lrmscalc*)

This subroutine actually handles the spherical harmonic transforms from (theta,phi) space to (ell,m) space.

Parameters

- **this** *[real]*
- **nr** *[integer,in]*
- **lrmscalc** *[logical,in]*

Call to *get_openmp_blocks()*, *spat_to_qst()*, *scal_to_sh()*, *spat_to_sphertor()*, *transform_to_lm_rms()*

10.9.4 get_nl.f90

Types

- **type** general_arrays_mod/**unknown_type**

10.9.5 get_td.f90

Description

This module is used to finish the assembly of the nonlinear terms in (ℓ, m) space. Derivatives along θ and ϕ are handled using recurrence relations.

Quick access

Routines *get_td()*, *set_zero()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *truncation* (*lm_max()*, *l_max()*, *lm_maxmag()*, *lmp_max()*): This module defines the grid points and the truncation
- *logic* (*l_anel()*, *l_conv_nl()*, *l_corr()*, *l_heat()*, *l_anelastic_liquid()*, *l_mag_nl()*, *l_mag_kin()*, *l_mag_lf()*, *l_conv()*, *l_mag()*, *l_chemical_conv()*, *l_single_matrix()*, *l_double_curl()*, *l_adv_curl()*, *l_phase_field()*): Module containing the logicals that control the run

- *radial_functions* (*r()*, *or2()*, *or1()*, *beta()*, *epscprof()*, *or4()*, *temp0()*, *orho1()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *physical_parameters* (*corfac()*, *epsc()*, *n_r_lcr()*, *epscxi()*): Module containing the physical parameters
- *blocking* (*lm2l()*, *lm2m()*, *lm2lmp()*, *lmp2lmps()*, *lmp2lmpa()*, *lm2lma()*, *lm2lms()*): Module containing blocking information
- *horizontal_data* (*dlh()*, *dphi()*, *dtheta2a()*, *dtheta3a()*, *dtheta4a()*, *dtheta2s()*, *dtheta3s()*, *dtheta4s()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *constants* (*zero()*, *two()*): module containing constants and parameters used in the code.
- *fields* (*w_rloc()*, *dw_rloc()*, *ddw_rloc()*, *z_rloc()*, *dz_rloc()*): This module contains all the fields used in MagIC in the hybrid (LM,r) space as well as their radial derivatives. It defines both the LM-distributed arrays and the R-distributed arrays....

Types

- **type** nonlinear_lm_mod/**unknown_type**

  Type fields

  - **% advplm** (*) *[complex,allocatable]*
  - **% advrlm** (*) *[complex,allocatable]*
  - **% advtlm** (*) *[complex,allocatable]*
  - **% dphidtlm** (*) *[complex,allocatable]*
  - **% heattermslm** (*) *[complex,allocatable]*
  - **% vsplm** (*) *[complex,allocatable]*
  - **% vsrlm** (*) *[complex,allocatable]*
  - **% vstlm** (*) *[complex,allocatable]*
  - **% vxbplm** (*) *[complex,allocatable]*
  - **% vxbrlm** (*) *[complex,allocatable]*
  - **% vxbtlm** (*) *[complex,allocatable]*
  - **% vxiplm** (*) *[complex,allocatable]*
  - **% vxirlm** (*) *[complex,allocatable]*
  - **% vxitlm** (*) *[complex,allocatable]*

Subroutines and functions

subroutine nonlinear_lm_mod/**initialize**(*this*, *lmp_max*)

Memory allocation of `get_td` arrays

Parameters

- **this** *[real]*
- **lmp_max** *[integer,in]*

subroutine nonlinear_lm_mod/**finalize**(*this*)

Memory deallocation

Parameters **this** *[real]*

subroutine nonlinear_lm_mod/**set_zero**(*this*)

Set all the arrays to zero

Parameters **this** *[real]*

subroutine nonlinear_lm_mod/**get_td**(*this*, *nr*, *nbc*, *lpressnext*, *dvsrlm*, *dvxirlm*, *dvxvhlm*, *dvxbhlm*, *dwdt*, *dzdt*, *dpdt*, *dsdt*, *dxidt*, *dphidt*, *dbdt*, *djdt*)

Purpose of this to calculate time derivatives `dwdt`,``dzdt``,``dpdt``,``dsdt``,``dxidt``,``dbdt``,``djdt`` and auxiliary arrays `dVSrLM`, `dVXirLM` and `dVxBhLM`, `dVxVhLM` from non-linear terms in spectral form

Parameters

- **this** *[real]*
- **nr** *[integer,in]* :: Radial level
- **nbc** *[integer,in]* :: signifies boundary conditions
- **lpressnext** *[logical,in]*
- **dvsrlm** (*) *[complex,out]*
- **dvxirlm** (*) *[complex,out]*
- **dvxvhlm** (*) *[complex,out]*
- **dvxbhlm** (*) *[complex,out]*
- **dwdt** (*) *[complex,out]* :: +CorPol_loc
- **dzdt** (*) *[complex,out]*
- **dpdt** (*) *[complex,out]*
- **dsdt** (*) *[complex,out]*
- **dxidt** (*) *[complex,out]*
- **dphidt** (*) *[complex,out]*
- **dbdt** (*) *[complex,out]*
- **djdt** (*) *[complex,out]*

10.9.6 nonlinear_bcs.f90

Quick access

Routines *get_b_nl_bcs()*, *get_br_v_bcs()*, *v_rigid_boundary()*

Needed modules

- iso_fortran_env (output_unit())
- *precision_mod*: This module controls the precision used in MagIC
- *truncation* (*lmp_max()*, *n_phi_max()*, *l_axi()*, *l_max()*, *n_theta_max()*, *nlat_padded()*): This module defines the grid points and the truncation
- *radial_data* (*n_r_cmb()*, *n_r_icb()*): This module defines the MPI decomposition in the radial direction.
- *radial_functions* (*r_cmb()*, *r_icb()*, *rho0()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *blocking* (*lm2l()*, *lm2m()*, *lm2lmp()*, *lmp2lmps()*, *lmp2lmpa()*): Module containing blocking information
- *physical_parameters* (*sigma_ratio()*, *conductance_ma()*, *prmag()*, *oek()*): Module containing the physical parameters
- *horizontal_data* (*dtheta1s()*, *dtheta1a()*, *dphi()*, *o_sin_theta_e2()*, *dlh()*, *sn2()*, *costheta()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *constants* (*two()*): module containing constants and parameters used in the code.
- *useful* (*abortrun()*): This module contains several useful routines.
- *sht* (*scal_to_sh()*)

Variables

Subroutines and functions

subroutine nonlinear_bcs/**get_br_v_bcs**(*br*, *vt*, *vp*, *omega*, *o_r_e_2*, *o_rho*, *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** (,) *[real,in]* :: $r^2 B_r$
- **vt** (,) *[real,in]* :: $r \sin\theta u_\theta$
- **vp** (,) *[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]* :: $1/\tilde{\rho}$ (anelastic)
- **br_vt_lm** (*lmp_max*) *[complex,inout]*
- **br_vp_lm** (*lmp_max*) *[complex,inout]*

Called from *radialloop()*

Call to *scal_to_sh()*

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*)

Purpose of this subroutine is to calculate the nonlinear term of the magnetic boundary condition for a conducting mantle in physical space (theta,phi), assuming that the conductance of the mantle is much smaller than that of the core. Calculation is performed for the theta block:

```
n_theta_min<=n_theta<=n_theta_min+n_theta_block-1
```

Parameters

- **bc** *[character,in]* :: Distinguishes 'CMB' and 'ICB'
- **br_vt_lm** (*lmp_max*) *[complex,in]* :: $B_r u_\theta/(r^2 \sin^2\theta)$
- **br_vp_lm** (*lmp_max*) *[complex,in]* :: $B_r u_\phi/(r^2 \sin^2\theta)$
- **lm_min_b** *[integer,in]*
- **lm_max_b** *[integer,in]* :: limits of lm-block
- **b_nl_bc** (1 + lm_max_b - lm_min_b) *[complex,out]* :: nonlinear bc for b
- **aj_nl_bc** (1 + lm_max_b - lm_min_b) *[complex,out]* :: nonlinear bc for aj

Called from *step_time()*

Call to *abortrun()*

subroutine nonlinear_bcs/**v_rigid_boundary**(*nr*, *omega*, *lderiv*, *vrr*, *vtr*, *vpr*, *cvrr*, *dvrdtr*, *dvrdpr*, *dvtdpr*, *dvpdpr*)

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-axis, `vp= r \sin(theta) v_phi = r**2 sin(theta)**2 omega` and `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]* :: boundary rotation rate
- **lderiv** *[logical,in]* :: derivatives required ?
- **vrr** (,) *[real,out]*
- **vtr** (,) *[real,out]*
- **vpr** (,) *[real,out]*
- **cvrr** (,) *[real,out]*
- **dvrdtr** (,) *[real,out]*
- **dvrdpr** (,) *[real,out]*

- **dvtdpr** (,) *[real,out]*
- **dvpdpr** (,) *[real,out]*

Called from *transform_to_grid_space()*

10.10 Radial scheme

10.10.1 radial_scheme.f90

Description

This is an abstract type that defines the radial scheme used in MagIC

Quick access

Routines *costf1_complex()*, *costf1_complex_1d()*, *costf1_real_1d()*

Needed modules

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

Types

- **type** radial_scheme/**unknown_type**

Type fields

- **% boundary_fac** *[real]*
- **% d2rmat** (,) *[real,allocatable]*
- **% d3rmat** (,) *[real,allocatable]*
- **% d4rmat** (,) *[real,allocatable]*
- **% ddddr** (,) *[real,allocatable]*
- **% dddr** (,) *[real,allocatable]*
- **% dddr_bot** (,) *[real,allocatable]*
- **% dddr_top** (,) *[real,allocatable]*
- **% dddrx** (*) *[real,allocatable]*
- **% ddr** (,) *[real,allocatable]*
- **% ddr_bot** (,) *[real,allocatable]*
- **% ddr_top** (,) *[real,allocatable]*
- **% ddrx** (*) *[real,allocatable]*
- **% dr** (,) *[real,allocatable]*
- **% dr_bot** (,) *[real,allocatable]*
- **% dr_top** (,) *[real,allocatable]*

- **% drmat** (,) *[real,allocatable]*
- **% drx** (*) *[real,allocatable]*
- **% n_max** *[integer]*
- **% nrmax** *[integer]*
- **% order** *[integer]*
- **% order_boundary** *[integer]*
- **% rmat** (,) *[real,allocatable]*
- **% rnorm** *[real]*
- **% version** *[character]*

Subroutines and functions

subroutine radial_scheme/**costf1_complex**(*this*, *f*, *n_f_max*, *n_f_start*, *n_f_stop*[, *work_array*])

Parameters

- **this** *[real]*
- **f** (n_f_max,this%nrmax) *[complex,inout]*
- **n_f_max** *[integer,in,]* :: number of columns in f,f2
- **n_f_start** *[integer,in]*
- **n_f_stop** *[integer,in]* :: columns to be transformed
- **work_array** (n_f_max,this%nrmax) *[complex,inout,target]*

subroutine radial_scheme/**costf1_real_1d**(*this*, *f*)

Parameters

- **this** *[real]*
- **f** (this%nrmax) *[real,inout]*

subroutine radial_scheme/**costf1_complex_1d**(*this*, *f*)

Parameters

- **this** *[real]*
- **f** (this%nrmax) *[complex,inout]*

10.10.2 chebyshev.f90

Quick access

Routines *initialize_mapping()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *constants* (*half()*, *one()*, *two()*, *three()*, *four()*, *pi()*): module containing constants and parameters used in the code.
- *blocking* (*llm()*, *ulm()*): Module containing blocking information
- *radial_scheme* (`type_rscheme()`): This is an abstract type that defines the radial scheme used in MagIC
- *useful* (*factorise()*): This module contains several useful routines.
- *chebyshev_polynoms_mod* (*cheb_grid()*)
- *cosine_transform_odd* (`costf_odd_t()`): This module contains the built-in type I discrete Cosine Transforms. This implementation is based on Numerical Recipes and FFTPACK. This only works for `n_r_max-1 = 2**a 3**b 5**c`, with a,b,c integers. ...
- *num_param* (*map_function()*): Module containing numerical and control parameters

Types

- **type** chebyshev/**unknown_type**

  Type fields

  - **% alpha1** *[real]*
  - **% alpha2** *[real]*
  - **% chebt_oc** *[costf_odd_t]*
  - **% l_map** *[logical]*
  - **% r_cheb** (*) *[real,allocatable]*
  - **% work_costf** (,) *[complex,pointer]*

Subroutines and functions

subroutine chebyshev/**initialize**(*this*, *n_r_max*, *order*, *order_boundary*)

Purpose of this subroutine is to calculate and store several values that will be needed for a fast cosine transform of the first kind. The actual transform is performed by the subroutine `costf1`.

Parameters

- **this** *[real]*
- **n_r_max** *[integer,in]*

- **order** *[integer,in]*
- **order_boundary** *[integer,in]*

subroutine chebyshev/**initialize_mapping**(*this*, *n_r_max*, *ricb*, *rcmb*, *ratio1*, *ratio2*, *r*)

Parameters

- **this** *[real]*
- **n_r_max** *[integer,in]*
- **ricb** *[real,in]*
- **rcmb** *[real,in]*
- **ratio1** *[real,inout]*
- **ratio2** *[real,in]*
- **r** (*n_r_max*) *[real,out]*

Call to *cheb_grid()*

subroutine chebyshev/**finalize**(*this*)

Parameters **this** *[real]*

subroutine chebyshev/**robin_bc**(*this*, *atop*, *btop*, *rhs_top*, *abot*, *bbot*, *rhs_bot*, *f*)

This subroutine is used to determine the two boundary points of a field f subject to two Robin boundary conditions of the form:

```
atop*df/dr+btop*f = rhs_top;  abot*df/dr+bbot*f = rhs_bot
```

The method follows Canuto, SIAM, 1986 (p. 818)

Parameters

- **this** *[real]*
- **atop** *[real,in]*
- **btop** *[real,in]*
- **rhs_top** *[complex,in]*
- **abot** *[real,in]*
- **bbot** *[real,in]*
- **rhs_bot** *[complex,in]*
- **f** (*) *[complex,inout]*

subroutine chebyshev/**get_der_mat**(*this*, *n_r_max*)

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 polynoms are only defined in $[-1, 1]$ we have to use a map, mapping the points x to the 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

- **this** *[real]*
- **n_r_max** *[integer,in]*

subroutine chebyshev/**costf1_complex**(*this*, *f*, *n_f_max*, *n_f_start*, *n_f_stop*[, *work_array*])

Purpose of this subroutine is to perform a multiple cosine transforms for n+1 datapoints on the columns numbered n_f_start to n_f_stop in the array `f(n_f_max,n+1)` Depending whether the input f contains data or coeff arrays coeffs or data are returned in f.

Parameters

- **this** *[real]*
- **f** (n_f_max,this%nrmax) *[complex,inout]*
- **n_f_max** *[integer,in,]*
- **n_f_start** *[integer,in]*
- **n_f_stop** *[integer,in]*
- **work_array** (n_f_max,this%nrmax) *[complex,inout,target]*

subroutine chebyshev/**costf1_complex_1d**(*this*, *f*)

Parameters

- **this** *[real]*
- **f** (this%nrmax) *[complex,inout]*

subroutine chebyshev/**costf1_real_1d**(*this*, *f*)

Parameters

- **this** *[real]*
- **f** (this%nrmax) *[real,inout]*

10.10.3 finite_differences.f90

Description

This module is used to calculate the radial grid when finite differences are requested

Quick access

Routines *get_der_mat()*, *get_fd_coeffs()*, *get_fd_grid()*, *populate_fd_weights()*, *robin_bc()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *constants* (*zero()*, *one()*, *two()*, *half()*): module containing constants and parameters used in the code.
- *useful* (*logwrite()*, *abortrun()*): This module contains several useful routines.
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *radial_scheme* (`type_rscheme()`): This is an abstract type that defines the radial scheme used in MagIC

Types

- **type** finite_differences/**unknown_type**

  Type fields

  - **% ddddr_bot** (,) *[real,allocatable]*
  - **% ddddr_top** (,) *[real,allocatable]*

Subroutines and functions

subroutine finite_differences/**initialize**(*this*, *n_r_max*, *order*, *order_boundary*)

This subroutine allocates the arrays used when finite difference are used

Parameters

- **this** *[real]*
- **n_r_max** *[integer,in]*
- **order** *[integer,in]*
- **order_boundary** *[integer,in]*

subroutine finite_differences/**finalize**(*this*)

This subroutine deallocates the arrays used in FD

Parameters **this** *[real]*

subroutine finite_differences/**get_fd_grid**(*this*, *n_r_max*, *ricb*, *rcmb*, *ratio1*, *ratio2*, *r*)

This subroutine constructs the radial grid

Parameters

- **this** *[real]*
- **n_r_max** *[integer,in]* :: Number of grid points
- **ricb** *[real,in]* :: inner boundary
- **rcmb** *[real,in]* :: start with the outer boundary
- **ratio1** *[real,inout]* :: Nboudary/Nbulk

- **ratio2** *[real,in]* :: drMin/drMax
- **r** (*n_r_max*) *[real,out]* :: radius

Call to *logwrite()*, *abortrun()*

subroutine finite_differences/**get_fd_coeffs**(*this*, *r*)

Parameters

- **this** *[real]*
- **r** (*) *[real,in]* :: Radius

Call to *populate_fd_weights()*

subroutine finite_differences/**robin_bc**(*this*, *atop*, *btop*, *rhs_top*, *abot*, *bbot*, *rhs_bot*, *f*)

This routine is used to derive the values of f at both boundaries when f is subject to Robin boundary conditions on both sides:

Parameters

- **atop** *[real,in]* :: df/dr + btop * f = rhs_top; abot * df/dr + bbot * f = rhs_bot

  With finite differences, this yields two uncoupled equations that can be solved sequentially.
- **this** *[real]*
- **btop** *[real,in]*
- **rhs_top** *[complex,in]*
- **abot** *[real,in]*
- **bbot** *[real,in]*
- **rhs_bot** *[complex,in]*
- **f** (*) *[complex,inout]* :: Field f

subroutine finite_differences/**get_der_mat**(*this*, *n_r_max*)

Parameters

- **this** *[real]*
- **n_r_max** *[integer,in]*

subroutine finite_differences/**populate_fd_weights**(*z*, *x*, *nd*, *m*, *c*)

Generation of Finite Difference Formulas on Arbitrarily Spaced Grids, Bengt Fornberg, Mathematics of compuation, 51, 184, 1988, 699-706

Parameters

- **z** *[real,in]* :: grid points where approximations are to be accurate
- **x** (1 + *nd*) *[real,in]* :: hrid point locations
- **m** *[integer,in]* :: highest deriative for which weights are sought
- **c** (1 + *nd*,1 + m) *[real,out]*

Options **nd** *[integer,in,optional/default=-1 + shape(x, 0)]* :: dimension of `x` and `c`

Called from *get_fd_coeffs()*

10.11 Chebyshev polynomials and cosine transforms

10.11.1 chebyshev_polynoms.f90

Quick access

Routines *cheb_grid()*, *get_chebs_even()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *constants* (*pi()*, *half()*, *one()*, *two()*, *four()*): module containing constants and parameters used in the code.
- *num_param* (*map_function()*): Module containing numerical and control parameters

Variables

Subroutines and functions

subroutine chebyshev_polynoms_mod/**get_chebs_even**(*n_r*, *a*, *b*, *y*, *n_r_max*, *cheb*, *dcheb*, *d2cheb*, *dim1*, *dim2*)

Construct even Chebyshev polynomials and their first, second and third derivative up to degree `2*(n_r/2)` at `(n_r/2)` points x in the interval $[a, b]$. Since the polynoms are only defined in $[-1, 1]$ we have to map the points x in [a,b] onto points y in the interval $[-1, 1]$. For even Chebs we need only half the point of the map, these `(n_r/2)` points are in the interval $[1, 0[$.

Parameters

- **n_r** *[integer,in]* :: only even chebs stored !
- **a** *[real,in]*
- **b** *[real,in]* :: interval boundaries $[a, b]$
- **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 Chebyshev pol.
- **dcheb** (dim1,dim2) *[real,out]* :: first derivative of cheb
- **d2cheb** (dim1,dim2) *[real,out]* :: second derivative o cheb
- **dim1** *[integer,in]*
- **dim2** *[integer,in]* :: dimensions of cheb,dcheb,.....

Called from *radial()*

subroutine chebyshev_polynoms_mod/**cheb_grid**(*a*, *b*, *n*, *x*, *y*, *a1*, *a2*, *x0*, *lbd*, *l_map*)

Given the interval $[a, b]$ the routine returns the n+1 points that should be used to support a Chebyshev expansion. These are the n+1 extrema `y(i)` of the Chebyshev 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]*
- **b** *[real,in]* :: interval boundaries
- **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]*
- **l_map** *[logical,in]* :: Chebyshev mapping

Called from *initialize_mapping()*, *radial()*

10.11.2 cosine_transform_odd.f90

Description

This module contains the built-in type I discrete Cosine Transforms. This implementation is based on Numerical Recipes and FFTPACK. This only works for `n_r_max-1 = 2**a 3**b 5**c`, with a,b,c integers.

Quick access

Routines *costf1_real()*

Needed modules

- `iso_fortran_env` (`output_unit()`)
- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *fft_fac_mod* (*fft_fac_complex()*, *fft_fac_real()*)
- *constants* (*half()*, *one()*, *two()*, *pi()*, *sin36()*, *cos36()*, *sin60()*, *sin72()*, *cos72()*): module containing constants and parameters used in the code.
- *useful* (*factorise()*, *abortrun()*): This module contains several useful routines.

Types

- **type** cosine_transform_odd/**unknown_type**

  Type fields

  - **% d_costf_init** (*) *[real,allocatable]*
  - **% i_costf_init** (*) *[integer,allocatable]*

Subroutines and functions

subroutine cosine_transform_odd/**initialize**(*this*, *n*, *ni*, *nd*)

Purpose of this subroutine is to calculate and store several values that will be needed for a fast cosine transform of the first kind. The actual transform is performed by the subroutine `costf1`.

Parameters

- **this** *[real]*
- **n** *[integer,in]*
- **ni** *[integer,in]*
- **nd** *[integer,in]*

Call to *abortrun()*, *factorise()*

subroutine cosine_transform_odd/**finalize**(*this*)

Memory deallocation of help arrays for built-in type I DCT's

Parameters **this** *[real]*

subroutine cosine_transform_odd/**costf1_complex**(*this*, *f*, *n_f_max*, *n_f_start*, *n_f_stop*, *f2*)

Purpose of this subroutine is to perform a multiple cosine transforms for n+1 datapoints on the columns numbered n_f_start to n_f_stop in the array `f(n_f_max,n+1)` Depending whether the input f contains data or coeff arrays coeffs or data are returned in f.

Parameters

- **this** *[real]*
- **f** (n_f_max,*) *[complex,inout]*
- **n_f_max** *[integer,in,]*
- **n_f_start** *[integer,in]*
- **n_f_stop** *[integer,in]*
- **f2** (n_f_max,*) *[complex,out]*

Call to *fft_fac_complex()*

subroutine cosine_transform_odd/**costf1_complex_1d**(*this*, *f*, *f2*)

Built-in type I DCT for one single complex vector field

Parameters

- **this** *[real]*
- **f** (*) *[complex,inout]*
- **f2** (*) *[complex,out]*

subroutine cosine_transform_odd/**costf1_real**(*this*, *f*, *n_f_max*, *n_f_start*, *n_f_stop*, *f2*)

Built-in type I DCT for many real vector fields

Parameters

- **this** *[real]*
- **f** (n_f_max,*) *[real,inout]* :: data/coeff input
- **n_f_max** *[integer,in,]* :: number of columns in f,f2
- **n_f_start** *[integer,in]*
- **n_f_stop** *[integer,in]* :: columns to be transformed
- **f2** (n_f_max,*) *[real,out]* :: work array of the same size as f

Call to *fft_fac_real()*

subroutine cosine_transform_odd/**costf1_real_1d**(*this*, *f*, *f2*)

Built-in type I DCT for one single real vector field

Parameters

- **this** *[real]*
- **f** (*) *[real,inout]*
- **f2** (*) *[real,out]*

10.11.3 cosine_transform_even.f90

Quick access

Routines *costf2()*

Needed modules

- iso_fortran_env (output_unit())
- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *truncation* (*lm_max()*): This module defines the grid points and the truncation
- *fft_fac_mod* (*fft_fac_complex()*)
- *constants* (*half()*, *one()*, *two()*, *pi()*, *sin36()*, *cos36()*, *sin60()*, *sin72()*, *cos72()*): module containing constants and parameters used in the code.

- *useful* (*factorise()*, *abortrun()*): This module contains several useful routines.

Types

- **type** cosine_transform_even/**unknown_type**

  Type fields

  - **% d_costf_init** (*) *[real,allocatable]*
  - **% i_costf_init** (*) *[integer,allocatable]*

Subroutines and functions

subroutine cosine_transform_even/**initialize**(*this*, *n*, *ni*, *nd*)

Purpose of this subroutine is to calculate several things needed for the Chebyshev transform. Prepares costf2 for even number of grid points.

Parameters

- **this** *[real]*
- **n** *[integer,in]*
- **ni** *[integer,in]*
- **nd** *[integer,in]*

Call to *abortrun()*, *factorise()*

subroutine cosine_transform_even/**finalize**(*this*)

Parameters **this** *[real]*

subroutine cosine_transform_even/**costf2**(*this*, *f*, *n_f_max*, *n_f_start*, *n_f_stop*, *f2*)

Purpose of this subroutine is to perform a multiple cosine transforms for n+1 datapoints on the columns numbered n_f_start to n_f_stop in the array y(n_f_max,n+1) Depending whether the input y contains data or coeff arrays coeffs or data are returned in y.

Parameters

- **this** *[real]*
- **f** (n_f_max,*) *[complex,inout]* :: data/coeff input
- **n_f_max** *[integer,in,]* :: number of columns in y,y2
- **n_f_start** *[integer,in]*
- **n_f_stop** *[integer,in]* :: columns to be transformed
- **f2** (n_f_max,*) *[complex,out]* :: work array of the same size as y

Call to *fft_fac_complex()*

10.11.4 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()*, *sin60()*, *sin72()*, *cos36()*, *cos72()*, *half()*): module containing constants and parameters used in the code.

Variables

Subroutines and functions

subroutine fft_fac_mod/**fft_fac_real**(*a*, *b*, *c*, *d*, *trigs*, *nv*, *l1*, *l2*, *n*, *ifac*, *la*)

Main part of Fourier / Chebyshev 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()*

subroutine fft_fac_mod/**fft_fac_complex**(*a*, *b*, *c*, *d*, *trigs*, *nv*, *l1*, *l2*, *n*, *ifac*, *la*)

Main part of Fourier / Chebyshev 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()*

10.12 Fourier transforms

10.12.1 fft.f90

Description

This module contains the native subroutines used to compute FFT's. They are based on the FFT99 package from Temperton: http://www.cesm.ucar.edu/models/cesm1.2/cesm/cesmBbrowser/html_code/cam/fft99.F90.html I simply got rid of the 'go to' and Fortran legacy statements Those transforms only work for prime decomposition that involve factors of 2, 3, 5

Quick access

Variables *d_fft_init*, *i_fft_init*, *nd*, *ni*

Routines *fft991()*, *fft99a()*, *fft99b()*, *fft_many()*, *fft_to_real()*, *finalize_fft()*, *ifft_many()*, *init_fft()*, *vpassm()*

Needed modules

- `iso_fortran_env` (`output_unit()`)
- *precision_mod*: This module controls the precision used in MagIC
- *useful* (*factorise()*, *abortrun()*): This module contains several useful routines.
- *constants* (*pi()*, *sin36()*, *sin60()*, *sin72()*, *cos36()*, *cos72()*, *one()*, *two()*, *half()*): module containing constants and parameters used in the code.
- *truncation* (*n_phi_max()*, *nlat_padded()*): This module defines the grid points and the truncation
- *parallel_mod*: This module contains the blocking information

Variables

- fft/**d_fft_init** (*) *[real,private/allocatable]*
- fft/**i_fft_init** (100) *[integer,private]*
- fft/**nd** *[integer,private]*
- fft/**ni** *[integer,private/parameter/optional/default=100]*

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 Fourier transforms.

Parameters **n** *[integer,in]* :: Dimension of problem, number of grid points

Called from *horizontal()*

Call to *abortrun()*, *factorise()*

subroutine fft/**finalize_fft**()

Memory deallocation of FFT help arrays

Called from *finalize_horizontal_data()*

subroutine fft/**fft_to_real**(*f*, *ld_f*, *nrep*)

Parameters

- **f** (ld_f,nrep) *[real,inout]*
- **ld_f** *[integer,in,]*
- **nrep** *[integer,in,]*

Call to *fft991()*

subroutine fft/**fft_many**(*g*, *f*)

Fourier transform: `f(nlat,nlon) -> fhat(nlat,nlon/2+1)`

Parameters

- **g** (,) *[real,in]*
- **f** (*nlat_padded*,1 + *n_phi_max* / 2) *[complex,out]*

Called from *native_spat_to_sph()*, *native_spat_to_sph_tor()*

Call to *get_openmp_blocks()*, *fft991()*

subroutine fft/**ifft_many**(*f*, *g*)

Inverse Fourier transform: `fhat(nlat, nlon/2+1) -> f(nlat,nlon)`

Parameters

- **f** (,) *[complex,in]*

- **g** (*nlat_padded*,*n_phi_max*) *[real,out]*

Called from *native_qst_to_spat()*, *native_sphtor_to_spat()*, *native_sph_to_spat()*, *native_sph_to_grad_spat()*

Call to *get_openmp_blocks()*, *fft991()*

subroutine fft/**fft991**(*a*, *work*, *trigs*, *ifax*, *inc*, *jump*, *n*, *lot*, *isign*)

Multiple real/half-complex periodic Fast Fourier Transform

Procedure used to convert to half-length complex transform is given by Cooley, Lewis and Welch (j. sound vib., vol. 12 (1970), 315-337)

Definition of transforms:

`isign=+1: x(j)=sum(k=0,...,n-1)(c(k)*exp(2*i*j*k*pi/n))` where `c(k)=a(k)+i*b(k)` and `c(n-k)=a(k)-i*b(k)`

`isign=-1: a(k)=(1/n)*sum(j=0,...,n-1)(x(j)*cos(2*j*k*pi/n))` `b(k)=-(1/n)*sum(j=0,...,n-1)(x(j)*sin(2*j*k*pi/n))`

Parameters

- **a** (*) *[real,inout]* :: array containing input and output data
- **work** (lot + lot * n) *[real,inout]*
- **trigs** (*) *[real,in]* :: previously prepared list of trig function values
- **ifax** (*) *[integer,inout]* :: previously prepared list of factors of n/2
- **inc** *[integer,in]* :: increment within each data vector
- **jump** *[integer,in]* :: increment between the start of each data vector
- **lot** *[integer,in,]* :: number of data vectors
- **isign** *[integer,in]* :: sign of the FFT

Options **n** *[integer,in,optional/default=(-lot + shape(work, 0)) / lot]* :: length of the data vectors

Called from *fft_to_real()*, *fft_many()*, *ifft_many()*

Call to *fft99a()*, *vpassm()*, *fft99b()*

subroutine fft/**fft99a**(*a*, *work*, *trigs*, *inc*, *jump*, *n*, *lot*)

Parameters

- **a** (*) *[real,inout]*
- **work** (*) *[real,inout]*
- **trigs** (*) *[real,in]*
- **inc** *[integer,in]*
- **jump** *[integer,in]*
- **n** *[integer,in]*
- **lot** *[integer,in]*

Called from *fft991()*

subroutine fft/**fft99b**(*work*, *a*, *trigs*, *inc*, *jump*, *n*, *lot*)

Parameters

- **work** (*) *[real,inout]*
- **a** (*) *[real,inout]*
- **trigs** (*) *[real,in]*
- **inc** *[integer,in]*
- **jump** *[integer,in]*
- **n** *[integer,in]*
- **lot** *[integer,in]*

Called from *fft991()*

subroutine fft/**vpassm**(*a*, *b*, *c*, *d*, *trigs*, *inc1*, *inc2*, *inc3*, *inc4*, *lot*, *n*, *ifac*, *la*)

Parameters

- **a** (*) *[real,in]*
- **b** (*) *[real,in]*
- **c** (*) *[real,out]*
- **d** (*) *[real,out]*
- **trigs** (*) *[real,in]*
- **inc1** *[integer,in]*
- **inc2** *[integer,in]*
- **inc3** *[integer,in]*
- **inc4** *[integer,in]*
- **lot** *[integer,in]*
- **n** *[integer,in]*
- **ifac** *[integer,in]*
- **la** *[integer,in]*

Called from *fft991()*

10.13 Spherical harmonic transforms

10.13.1 plms.f90

Quick access

Routines *plm_theta()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *constants* (*osq4pi()*, *one()*, *two()*): module containing constants and parameters used in the code.
- *useful* (*abortrun()*): This module contains several useful routines.

Variables

Subroutines and functions

subroutine plms_theta/**plm_theta**(*theta*, *max_degree*, *max_order*, *m0*, *plma*, *dtheta_plma*, *ndim_plma*, *norm*)

This produces the P_ℓ^m for all degrees ℓ and orders m for a given colatitude. θ, as well as $\sin\theta dP_\ell^m/d\theta$. The P_ℓ^m are stored with a single lm index stored with m first.

Several normalisation are supported:

- n=0 – surface normalised,
- n=1 – Schmidt normalised,
- n=2 – fully normalised.

Parameters

- **theta** *[real,in]* :: angle in radians
- **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]* :: associated Legendre polynomials 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()*, *initialize_magnetic_energy()*, *initialize_transforms()*

Call to *abortrun()*

10.13.2 shtransforms.f90

Quick access

Variables *d_mc2m*, *dplm*, *lstart*, *lstartp*, *lstop*, *lstopp*, *plm*, *wdplm*, *wplm*

Routines *finalize_transforms()*, *initialize_transforms()*, *native_axi_to_spat()*, *native_qst_to_spat()*, *native_spat_to_sh_axi()*, *native_spat_to_sph()*, *native_spat_to_sph_tor()*, *native_sph_to_grad_spat()*, *native_sph_to_spat()*, *native_sphtor_to_spat()*, *native_toraxi_to_spat()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *truncation* (*lm_max()*, *n_m_max()*, *l_max()*, *l_axi()*, *n_theta_max()*, *minc()*, *n_phi_max()*, *lmp_max()*, *m_max()*): This module defines the grid points and the truncation
- *blocking* (*lmp2l()*, *lmp2lm()*): Module containing blocking information
- *horizontal_data* (*gauleg()*, *o_sin_theta_e2()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *plms_theta* (*plm_theta()*)
- *constants* (*zero()*, *half()*, *one()*, *ci()*, *pi()*, *two()*): module containing constants and parameters used in the code.
- *fft* (*fft_many()*, *ifft_many()*): This module contains the native subroutines used to compute FFT's. They are based on the FFT99 package from Temperton: http://www.cesm.ucar.edu/models/cesm1.2/cesm/cesmBbrowser/html_code/cam/fft99.F90.html
- *parallel_mod* (*get_openmp_blocks()*): This module contains the blocking information

Variables

- shtransforms/**d_mc2m** (*) *[real,allocatable/public]*
- shtransforms/**dplm** (*,*) *[real,allocatable]*
- shtransforms/**lstart** (*) *[integer,allocatable]*
- shtransforms/**lstartp** (*) *[integer,allocatable]*
- shtransforms/**lstop** (*) *[integer,allocatable]*
- shtransforms/**lstopp** (*) *[integer,allocatable]*
- shtransforms/**plm** (*,*) *[real,allocatable]*
- shtransforms/**wdplm** (*,*) *[real,allocatable]*
- shtransforms/**wplm** (*,*) *[real,allocatable]*

Subroutines and functions

subroutine shtransforms/**initialize_transforms**()

> **Called from** *initialize_sht()*
>
> **Call to** *gauleg()*, *plm_theta()*

subroutine shtransforms/**finalize_transforms**()

> **Called from** *finalize_sht()*

subroutine shtransforms/**native_qst_to_spat**(*qlm*, *slm*, *tlm*, *brc*, *btc*, *bpc*, *lcut*)

> Vector spherical harmonic transform: take Q,S,T and transform them to a vector field

Parameters

- **qlm** (*lm_max*) *[complex,in]* :: Poloidal
- **slm** (*lm_max*) *[complex,in]* :: Spheroidal
- **tlm** (*lm_max*) *[complex,in]* :: Toroidal
- **brc** (*n_theta_max*,*n_phi_max*) *[real,out]*
- **btc** (*n_theta_max*,*n_phi_max*) *[real,out]*
- **bpc** (*n_theta_max*,*n_phi_max*) *[real,out]*
- **lcut** *[integer,in]*

Called from `torpol_to_spat()`, `torpol_to_curl_spat_ic()`, `torpol_to_spat_ic()`, `torpol_to_curl_spat()`

Call to `get_openmp_blocks()`, `ifft_many()`

subroutine shtransforms/**native_sphtor_to_spat**(*slm*, *tlm*, *btc*, *bpc*, *lcut*)

Use spheroidal and toroidal potentials to transform them to angular vector components btheta and bphi

Parameters

- **slm** (*lm_max*) *[complex,in]*
- **tlm** (*lm_max*) *[complex,in]*
- **btc** (*n_theta_max*,*n_phi_max*) *[real,out]*
- **bpc** (*n_theta_max*,*n_phi_max*) *[real,out]*
- **lcut** *[integer,in]*

Called from `sphtor_to_spat()`, `torpol_to_dphspat()`

Call to `get_openmp_blocks()`, `ifft_many()`

subroutine shtransforms/**native_axi_to_spat**(*slm*, *sc*)

Parameters

- **slm** (1 + *l_max*) *[complex,in]*
- **sc** (*n_theta_max*) *[real,out]*

Called from `axi_to_spat()`

subroutine shtransforms/**native_toraxi_to_spat**(*tlm*, *btc*, *bpc*)

Use spheroidal and toroidal potentials to transform them to angular vector components btheta and bphi

Parameters

- **tlm** (1 + *l_max*) *[complex,in]*
- **btc** (*n_theta_max*) *[real,out]*
- **bpc** (*n_theta_max*) *[real,out]*

Called from *toraxi_to_spat()*

subroutine shtransforms/**native_sph_to_spat**(*slm*, *sc*, *lcut*)

Spherical Harmonic Transform for a scalar input field

Parameters

- **slm** (*lm_max*) *[complex,in]*
- **sc** (*n_theta_max*,*n_phi_max*) *[real,out]*
- **lcut** *[integer,in]*

Called from *scal_to_spat()*, *pol_to_curlr_spat()*

Call to *get_openmp_blocks()*, *ifft_many()*

subroutine shtransforms/**native_sph_to_grad_spat**(*slm*, *gradtc*, *gradpc*, *lcut*)

Transform `s(l)` into `dsdt(theta)` and `dsdp(theta)`

Parameters

- **slm** (*lm_max*) *[complex,in]*
- **gradtc** (*n_theta_max*,*n_phi_max*) *[real,out]*
- **gradpc** (*n_theta_max*,*n_phi_max*) *[real,out]*
- **lcut** *[integer,in]*

Called from *scal_to_grad_spat()*, *pol_to_grad_spat()*

Call to *get_openmp_blocks()*, *ifft_many()*

subroutine shtransforms/**native_spat_to_sph**(*scal*, *f1lm*, *lcut*)

Legendre transform (n_r,n_theta,m) to (n_r,l,m)

Parameters

- **scal** (,) *[real,inout]*
- **f1lm** (*lmp_max*) *[complex,out]*
- **lcut** *[integer,in]*

Called from *scal_to_sh()*, *spat_to_qst()*

Call to *fft_many()*, *get_openmp_blocks()*

subroutine shtransforms/**native_spat_to_sph_tor**(*vt*, *vp*, *f1lm*, *f2lm*, *lcut*)

Vector Legendre transform `vt(n_r,n_theta,m)`, `vp(n_r,n_theta,m)` to `Spheroidal(n_r,l, m)` and `Toroidal(n_r,l,m)`

Parameters

- **vt** (,) *[real,inout]*
- **vp** (,) *[real,inout]*

- **f1lm** (*lmp_max*) *[complex,out]*
- **f2lm** (*lmp_max*) *[complex,out]*
- **lcut** *[integer,in]*

Called from *spat_to_qst()*, *spat_to_sphertor()*

Call to *fft_many()*, *get_openmp_blocks()*

subroutine shtransforms/**native_spat_to_sh_axi**(*ft1*, *flm1*, *lmmax*)

Legendre transform for an axisymmetric field

Parameters

- **ft1** (*) *[real,in]* :: ASymm
- **flm1** (*) *[real,out]*
- **lmmax** *[integer,in]* :: Number of modes to be processed

Called from *spat_to_sh_axi()*

10.13.3 sht_native.f90

Quick access

Routines *axi_to_spat()*, *finalize_sht()*, *initialize_sht()*, *pol_to_curlr_spat()*, *pol_to_grad_spat()*, *scal_to_grad_spat()*, *scal_to_sh()*, *scal_to_spat()*, *spat_to_qst()*, *spat_to_sh_axi()*, *spat_to_sphertor()*, *sphtor_to_spat()*, *toraxi_to_spat()*, *torpol_to_curl_spat()*, *torpol_to_curl_spat_ic()*, *torpol_to_dphspat()*, *torpol_to_spat()*, *torpol_to_spat_ic()*

Needed modules

- iso_fortran_env (output_unit())
- *precision_mod* (*cp()*): This module controls the precision used in MagIC
- *blocking* (*st_map()*): Module containing blocking information
- *constants* (*ci()*, *one()*, *zero()*): module containing constants and parameters used in the code.
- *truncation* (*m_max()*, *l_max()*, *n_theta_max()*, *n_phi_max()*, *minc()*, *lm_max()*, *lmp_max()*): This module defines the grid points and the truncation
- *horizontal_data* (*dlh()*, *o_sin_theta_e2()*, *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
- *shtransforms*

Variables

Subroutines and functions

subroutine sht/**initialize_sht**()

Called from *magic*

Call to *initialize_transforms()*

subroutine sht/**finalize_sht**()

Called from *magic*

Call to *finalize_transforms()*

subroutine sht/**scal_to_spat**(*slm*, *fieldc*, *lcut*)
transform a spherical harmonic field into grid space

Parameters

- **slm** (*) *[complex,in]*
- **fieldc** (,) *[real,out]*
- **lcut** *[integer,in]*

Called from *fields_average()*, *lm2pt()*, *transform_to_grid_space()*

Call to *native_sph_to_spat()*

subroutine sht/**scal_to_grad_spat**(*slm*, *gradtc*, *gradpc*, *lcut*)
transform a scalar spherical harmonic field into it's gradient on the grid

Parameters

- **slm** (*) *[complex,in]*
- **gradtc** (,) *[real,out]*
- **gradpc** (,) *[real,out]*
- **lcut** *[integer,in]*

Called from *transform_to_grid_rms()*, *transform_to_grid_space()*

Call to *native_sph_to_grad_spat()*

subroutine sht/**pol_to_grad_spat**(*slm*, *gradtc*, *gradpc*, *lcut*)

Parameters

- **slm** (*) *[complex,in]*
- **gradtc** (,) *[real,out]*
- **gradpc** (,) *[real,out]*
- **lcut** *[integer,in]*

Called from *transform_to_grid_space()*

Call to *native_sph_to_grad_spat()*

subroutine sht/**torpol_to_spat**(*wlm*, *dwlm*, *zlm*, *vrc*, *vtc*, *vpc*, *lcut*)

Parameters

- **wlm** (*) *[complex,in]*
- **dwlm** (*) *[complex,in]*
- **zlm** (*) *[complex,in]*
- **vrc** (,) *[real,out]*
- **vtc** (,) *[real,out]*
- **vpc** (,) *[real,out]*
- **lcut** *[integer,in]*

Called from *fields_average()*, *get_bpol()*, *get_b_surface()*, *transform_to_grid_space()*

Call to *native_qst_to_spat()*

subroutine sht/**sphtor_to_spat**(*dwlm*, *zlm*, *vtc*, *vpc*, *lcut*)

Parameters

- **dwlm** (*) *[complex,in]*
- **zlm** (*) *[complex,in]*
- **vtc** (,) *[real,out]*
- **vpc** (,) *[real,out]*
- **lcut** *[integer,in]*

Called from *get_btor()*

Call to *native_sphtor_to_spat()*

subroutine sht/**torpol_to_curl_spat_ic**(*r*, *r_icb*, *dblm*, *ddblm*, *jlm*, *djlm*, *cbr*, *cbt*, *cbp*)

This is a QST transform that contains the transform for the inner core to compute the three components of the curl of B.

Parameters

- **r** *[real,in]*
- **r_icb** *[real,in]*
- **dblm** (*) *[complex,in]*
- **ddblm** (*) *[complex,in]*
- **jlm** (*) *[complex,in]*
- **djlm** (*) *[complex,in]*
- **cbr** (,) *[real,out]*
- **cbt** (,) *[real,out]*
- **cbp** (,) *[real,out]*

Called from *store_movie_frame_ic()*

Call to *native_qst_to_spat()*

subroutine sht/**torpol_to_spat_ic**(*r*, *r_icb*, *wlm*, *dwlm*, *zlm*, *br*, *bt*, *bp*)

This is a QST transform that contains the transform for the inner core.

Parameters

- **r** *[real,in]*
- **r_icb** *[real,in]*
- **wlm** (*) *[complex,in]*
- **dwlm** (*) *[complex,in]*
- **zlm** (*) *[complex,in]*
- **br** (,) *[real,out]*
- **bt** (,) *[real,out]*
- **bp** (,) *[real,out]*

Called from *graphout_ic()*, *store_movie_frame_ic()*

Call to *native_qst_to_spat()*

subroutine sht/**torpol_to_dphspat**(*dwlm*, *zlm*, *dvtdp*, *dvpdp*, *lcut*)

Computes horizontal phi derivative of a toroidal/poloidal field

Parameters

- **dwlm** (*) *[complex,in]*
- **zlm** (*) *[complex,in]*
- **dvtdp** (,) *[real,out]*
- **dvpdp** (,) *[real,out]*
- **lcut** *[integer,in]*

Called from *transform_to_grid_space()*

Call to *native_sphtor_to_spat()*

subroutine sht/**pol_to_curlr_spat**(*qlm*, *cvrc*, *lcut*)

Parameters

- **qlm** (*) *[complex,in]*
- **cvrc** (,) *[real,out]*
- **lcut** *[integer,in]*

Called from *transform_to_grid_space()*

Call to *native_sph_to_spat()*

subroutine sht/**torpol_to_curl_spat**(*or2*, *blm*, *ddblm*, *jlm*, *djlm*, *cvrc*, *cvtc*, *cvpc*, *lcut*)
– Input variables

Parameters

- **or2** *[real,in]*

- **blm** (*) *[complex,in]*
- **ddblm** (*) *[complex,in]*
- **jlm** (*) *[complex,in]*
- **djlm** (*) *[complex,in]*
- **cvrc** (,) *[real,out]*
- **cvtc** (,) *[real,out]*
- **cvpc** (,) *[real,out]*
- **lcut** *[integer,in]*

Called from *transform_to_grid_space()*

Call to *native_qst_to_spat()*

subroutine sht/**scal_to_sh**(*f*, *flm*, *lcut*)

Parameters

- **f** (,) *[real,inout]*
- **flm** (*) *[complex,out]*
- **lcut** *[integer,in]*

Called from *transform_to_lm_rms()*, *get_dtblm()*, *initv()*, *inits()*, *initxi()*, *get_br_v_bcs()*, *transform_to_lm_space()*

Call to *native_spat_to_sph()*

subroutine sht/**spat_to_qst**(*f*, *g*, *h*, *qlm*, *slm*, *tlm*, *lcut*)

Parameters

- **f** (,) *[real,inout]*
- **g** (,) *[real,inout]*
- **h** (,) *[real,inout]*
- **qlm** (*) *[complex,out]*
- **slm** (*) *[complex,out]*
- **tlm** (*) *[complex,out]*
- **lcut** *[integer,in]*

Called from *transform_to_lm_rms()*, *transform_to_lm_space()*

Call to *native_spat_to_sph()*, *native_spat_to_sph_tor()*

subroutine sht/**spat_to_sphertor**(*f*, *g*, *flm*, *glm*, *lcut*)

Parameters

- **f** (,) *[real,inout]*
- **g** (,) *[real,inout]*
- **flm** (*) *[complex,out]*
- **glm** (*) *[complex,out]*

- **lcut** *[integer,in]*

Called from `transform_to_lm_rms()`, `transform_to_lm_space()`

Call to `native_spat_to_sph_tor()`

subroutine sht/**axi_to_spat**(*fl_ax*, *f*)

Parameters

- **fl_ax** (1 + *l_max*) *[complex,in]*
- **f** (*) *[real,out]*

Called from `outphase()`

Call to `native_axi_to_spat()`

subroutine sht/**toraxi_to_spat**(*fl_ax*, *ft*, *fp*)

Parameters

- **fl_ax** (1 + *l_max*) *[complex,in]*
- **ft** (*) *[real,out]*
- **fp** (*) *[real,out]*

Called from `get_pas()`, `outomega()`, `get_dtb()`, `get_sl()`, `get_fl()`

Call to `native_toraxi_to_spat()`

subroutine sht/**spat_to_sh_axi**(*f*, *flm*)

Parameters

- **f** (*n_theta_max*) *[real,in]*
- **flm** (*) *[real,out]*

Called from `getto()`

Call to `native_spat_to_sh_axi()`

10.14 Linear algebra

10.14.1 matrices.f90

Quick access

Routines `mat_add()`

Needed modules

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

Types

- **type** real_matrices/**unknown_type**

  Type fields

  - **% dat** (,) *[real,allocatable]*
  - **% l_pivot** *[logical]*
  - **% ncol** *[integer]*
  - **% nrow** *[integer]*
  - **% pivot** (*) *[integer,allocatable]*

Subroutines and functions

function real_matrices/**mat_add**(*this*, *b*)

Parameters

- **this** *[real]*
- **b** *[real]*

Return **mat_add** *[integer]*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc*: This little module is used to estimate the global memory allocation used in MagIC
- *real_matrices* (type_realmat())
- *algebra* (*solve_mat()*, *prepare_mat()*)

Types

- **type** dense_matrices/**unknown_type**

Subroutines and functions

subroutine dense_matrices/**initialize**(*this*, *nx*, *ny*, *l_pivot*[, *nfull*])

Memory allocation

Parameters

- **this** *[real]*
- **nx** *[integer,in]*
- **ny** *[integer,in]*
- **l_pivot** *[logical,in]*
- **nfull** *[integer,in,]*

subroutine dense_matrices/**finalize**(*this*)

Memory deallocation

Parameters **this** *[real]*

subroutine dense_matrices/**prepare**(*this*, *info*)

Parameters

- **this** *[real]*
- **info** *[integer,out]*

Call to *prepare_mat()*

subroutine dense_matrices/**solve_real_single**(*this*, *rhs*)

Parameters

- **this** *[real]*
- **rhs** (*) *[real,inout]*

subroutine dense_matrices/**solve_complex_single**(*this*, *rhs*)

Parameters

- **this** *[real]*
- **rhs** (*) *[complex,inout]*

subroutine dense_matrices/**solve_real_multi**(*this*, *rhs*, *nrhs*)

Parameters

- **this** *[real]*
- **rhs** (,) *[real,inout]*
- **nrhs** *[integer,in]*

subroutine dense_matrices/**set_data**(*this*, *dat*)

Parameters

- **this** *[real]*
- **dat** (,) *[real,in]*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc*: This little module is used to estimate the global memory allocation used in MagIC
- *real_matrices* (type_realmat())
- *algebra* (*solve_tridiag()*, *prepare_tridiag()*, *prepare_band()*, *solve_band()*)

Types

- **type** band_matrices/**unknown_type**

  Type fields

  - **% du2** (*) *[real,allocatable]*
  - **% kl** *[integer]*
  - **% ku** *[integer]*

Subroutines and functions

subroutine band_matrices/**initialize**(*this*, *nx*, *ny*, *l_pivot*[, *nfull*])

Memory allocation

Parameters

- **this** *[real]*
- **nx** *[integer,in]*
- **ny** *[integer,in]*
- **l_pivot** *[logical,in]*
- **nfull** *[integer,in,]*

subroutine band_matrices/**finalize**(*this*)

Memory deallocation

Parameters **this** *[real]*

subroutine band_matrices/**prepare**(*this*, *info*)

Parameters

- **this** *[real]*
- **info** *[integer,out]*

Call to *prepare_tridiag()*, *prepare_band()*

subroutine band_matrices/**solve_real_single**(*this*, *rhs*)

Parameters

- **this** *[real]*
- **rhs** (*) *[real,inout]*

subroutine band_matrices/**solve_complex_single**(*this*, *rhs*)

Parameters

- **this** *[real]*
- **rhs** (*) *[complex,inout]*

subroutine band_matrices/**solve_real_multi**(*this*, *rhs*, *nrhs*)

Parameters

- **this** *[real]*
- **rhs** (,) *[real,inout]*
- **nrhs** *[integer,in]*

subroutine band_matrices/**set_data**(*this*, *dat*)

Parameters

- **this** *[real]*
- **dat** (,) *[real,in]*

10.14.2 algebra.f90

Quick access

Variables *solve_band*, *solve_bordered*, *solve_mat*, *solve_tridiag*, *zero_tolerance*

Routines *gemm()*, *gemv()*, *prepare_band()*, *prepare_bordered()*, *prepare_mat()*, *prepare_tridiag()*, *solve_band_complex_rhs()*, *solve_band_real_rhs()*, *solve_band_real_rhs_multi()*, *solve_bordered_complex_rhs()*, *solve_bordered_real_rhs()*, *solve_bordered_real_rhs_multi()*, *solve_mat_complex_rhs()*, *solve_mat_real_rhs()*, *solve_mat_real_rhs_multi()*, *solve_tridiag_complex_rhs()*, *solve_tridiag_real_rhs()*, *solve_tridiag_real_rhs_multi()*

Needed modules

- omp_lib
- *precision_mod*: This module controls the precision used in MagIC
- *constants* (*one()*, *zero()*): module containing constants and parameters used in the code.
- *useful* (*abortrun()*): This module contains several useful routines.

Variables

- algebra/**solve_band** *[public]*
- algebra/**solve_bordered** *[public]*
- algebra/**solve_mat** *[public]*
- algebra/**solve_tridiag** *[public]*
- algebra/**zero_tolerance** *[real,private/parameter/optional/default=1.0e-15_cp]*

Subroutines and functions

subroutine algebra/**solve_mat_complex_rhs**(*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

subroutine algebra/**solve_mat_real_rhs_multi**(*a*, *ia*, *n*, *ip*, *bc*, *nrhss*)

This routine does the backward substitution into a LU-decomposed real matrix a (to solve a * x = bc) simultaneously for nRHSs real 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** (,) *[real,inout]* :: on input RHS of problem
- **nrhss** *[integer,in]* :: number of right-hand sides

Called from *solve_bordered_real_rhs_multi()*

subroutine algebra/**solve_mat_real_rhs**(*a*, *ia*, *n*, *ip*, *b*)

Backward substitution of vector b into lu-decomposed matrix a to solve a * x = b for a single real vector b

Parameters

- **a** (ia,*) *[real,in]* :: n*n real matrix
- **ia** *[integer,in,]* :: first dim of a
- **n** *[integer,in]* :: dim of problem
- **ip** (*) *[integer,in]* :: pivot information
- **b** (*) *[real,inout]* :: rhs-vector on input, solution on output

Called from *solve_bordered_real_rhs()*, *solve_bordered_complex_rhs()*

subroutine algebra/**prepare_mat**(*a*, *ia*, *n*, *ip*, *info*)

LU decomposition the real matrix a(n,n) via gaussian elimination

Parameters

- **a** (ia,*) *[real,inout]* :: real nxn matrix on input, lu-decomposed matrix on output
- **ia** *[integer,in,]* :: first dimension of a (must be >= n)
- **n** *[integer,in]* :: 2nd dimension and rank of a
- **ip** (*) *[integer,out]* :: pivot pointer array
- **info** *[integer,out]* :: error message when /= 0

Called from *prepare_bordered()*, *inits()*, *initxi()*, *j_cond()*, *pt_cond()*, *ps_cond()*, *getbackground()*, *get_wpsmat()*, *get_ps0mat()*

Call to *abortrun()*

subroutine algebra/**solve_band_real_rhs**(*abd*, *n*, *kl*, *ku*, *pivot*, *rhs*)

Parameters

- **abd** (1 + 2 * kl + ku,n) *[real,in]*
- **n** *[integer,in,]*
- **pivot** (n) *[integer,in]*
- **rhs** (n) *[real,inout]*

Options

- **kl** *[integer,in,optional/default=(-1 - ku + shape(abd, 0)) / 2]*
- **ku** *[integer,in,optional/default=-1 - 2 * kl + shape(abd, 0)]*

Called from *solve_bordered_real_rhs()*, *solve_bordered_complex_rhs()*

subroutine algebra/**solve_band_complex_rhs**(*abd*, *n*, *kl*, *ku*, *pivot*, *rhs*)

Parameters

- **abd** (1 + 2 * kl + ku,n) *[real,in]*

- **n** *[integer,in,]*
- **pivot** (n) *[integer,in]*
- **rhs** (n) *[complex,inout]*

Options

- **kl** *[integer,in,optional/default=(-1 - ku + shape(abd, 0)) / 2]*
- **ku** *[integer,in,optional/default=-1 - 2 * kl + shape(abd, 0)]*

subroutine algebra/**solve_band_real_rhs_multi**(*abd*, *n*, *kl*, *ku*, *pivot*, *rhs*, *nrhss*)

Parameters

- **abd** (1 + 2 * kl + ku,n) *[real,in]*
- **n** *[integer,in,]*
- **pivot** (n) *[integer,in]*
- **rhs** (,) *[real,inout]*
- **nrhss** *[integer,in]*

Options

- **kl** *[integer,in,optional/default=(-1 - ku + shape(abd, 0)) / 2]*
- **ku** *[integer,in,optional/default=-1 - 2 * kl + shape(abd, 0)]*

Called from *solve_bordered_real_rhs_multi()*, *prepare_bordered()*

subroutine algebra/**prepare_band**(*abd*, *n*, *kl*, *ku*, *pivot*, *info*)

Parameters

- **abd** (1 + 2 * kl + ku,n) *[real,inout]*
- **n** *[integer,in,]*
- **pivot** (n) *[integer,out]*
- **info** *[integer,out]*

Options

- **kl** *[integer,in,optional/default=(-1 - ku + shape(abd, 0)) / 2]*
- **ku** *[integer,in,optional/default=-1 - 2 * kl + shape(abd, 0)]*

Called from *prepare_bordered()*

subroutine algebra/**solve_tridiag_real_rhs**(*dl*, *d*, *du*, *du2*, *n*, *pivot*, *rhs*)

Parameters

- **dl** (*) *[real,in]* :: Lower
- **d** (*) *[real,in]* :: Diagonal
- **du** (*) *[real,in]* :: Upper
- **du2** (*) *[real,in]* :: For pivot
- **n** *[integer,in]* :: dim of problem

- **pivot** (*) *[integer,in]* :: pivot information
- **rhs** (*) *[real,inout]*

subroutine algebra/**solve_tridiag_complex_rhs**(*dl*, *d*, *du*, *du2*, *n*, *pivot*, *rhs*)

Parameters

- **dl** (*) *[real,in]* :: Lower
- **d** (*) *[real,in]* :: Diagonal
- **du** (*) *[real,in]* :: Lower
- **du2** (*) *[real,in]* :: Upper
- **n** *[integer,in]* :: dim of problem
- **pivot** (*) *[integer,in]* :: pivot information
- **rhs** (*) *[complex,inout]*

subroutine algebra/**solve_tridiag_real_rhs_multi**(*dl*, *d*, *du*, *du2*, *n*, *pivot*, *rhs*, *nrhss*)

Parameters

- **dl** (*) *[real,in]* :: Lower
- **d** (*) *[real,in]* :: Diagonal
- **du** (*) *[real,in]* :: Upper
- **du2** (*) *[real,in]* :: For pivot
- **n** *[integer,in]* :: dim of problem
- **pivot** (*) *[integer,in]* :: pivot information
- **rhs** (,) *[real,inout]*
- **nrhss** *[integer,in]* :: Number of right-hand side

subroutine algebra/**prepare_tridiag**(*dl*, *d*, *du*, *du2*, *n*, *pivot*, *info*)

Parameters

- **dl** (*) *[real,inout]*
- **d** (*) *[real,inout]*
- **du** (*) *[real,inout]*
- **du2** (*) *[real,out]*
- **n** *[integer,in]*
- **pivot** (*) *[integer,out]*
- **info** *[integer,out]*

subroutine algebra/**solve_bordered_real_rhs**(*a1*, *a2*, *a3*, *a4*, *lena1*, *n_boundaries*, *kl*, *ku*, *pivota1*, *pivota4*, *rhs*, *lenrhs*)

– Input variables

Parameters

- **a1** (1 + 2 * kl + ku,lena1) *[real,in]*
- **a2** (lena1,n_boundaries) *[real,in]*
- **a3** (lena1) *[real,in]*
- **a4** (n_boundaries,n_boundaries) *[real,in]*
- **lena1** *[integer,in,]*
- **n_boundaries** *[integer,in,]*
- **pivota1** (lena1) *[integer,in]*
- **pivota4** (n_boundaries) *[integer,in]*
- **rhs** (lenrhs) *[real,inout]*
- **lenrhs** *[integer,in,]*

Options

- **kl** *[integer,in,optional/default=(-1 - ku + shape(a1, 0)) / 2]*
- **ku** *[integer,in,optional/default=-1 - 2 * kl + shape(a1, 0)]*

Call to *solve_band_real_rhs()*, *solve_mat_real_rhs()*, *gemv()*

subroutine algebra/**solve_bordered_complex_rhs**(*a1*, *a2*, *a3*, *a4*, *lena1*, *n_boundaries*, *kl*, *ku*, *pivota1*, *pivota4*, *rhs*, *lenrhs*)

– Input variables

Parameters

- **a1** (1 + 2 * kl + ku,lena1) *[real,in]*
- **a2** (lena1,n_boundaries) *[real,in]*
- **a3** (lena1) *[real,in]*
- **a4** (n_boundaries,n_boundaries) *[real,in]*
- **lena1** *[integer,in,]*
- **n_boundaries** *[integer,in,]*
- **pivota1** (lena1) *[integer,in]*
- **pivota4** (n_boundaries) *[integer,in]*
- **rhs** (lenrhs) *[complex,inout]*
- **lenrhs** *[integer,in,]*

Options

- **kl** *[integer,in,optional/default=(-1 - ku + shape(a1, 0)) / 2]*
- **ku** *[integer,in,optional/default=-1 - 2 * kl + shape(a1, 0)]*

Call to *solve_band_real_rhs()*, *solve_mat_real_rhs()*, *gemv()*

subroutine algebra/**solve_bordered_real_rhs_multi**(*a1*, *a2*, *a3*, *a4*, *lena1*, *n_boundaries*, *kl*, *ku*, *pivota1*, *pivota4*, *rhs*, *nrhss*)

– Input variables

Parameters

- **a1** (1 + 2 * kl + ku,lena1) *[real,in]*

- **a2** (lena1,n_boundaries) *[real,in]*
- **a3** (lena1) *[real,in]*
- **a4** (n_boundaries,n_boundaries) *[real,in]*
- **lena1** *[integer,in,]*
- **n_boundaries** *[integer,in,]*
- **pivota1** (lena1) *[integer,in]*
- **pivota4** (n_boundaries) *[integer,in]*
- **rhs** (,) *[real,inout]*
- **nrhss** *[integer,in]*

Options

- **kl** *[integer,in,optional/default=(-1 - ku + shape(a1, 0)) / 2]*
- **ku** *[integer,in,optional/default=-1 - 2 * kl + shape(a1, 0)]*

Call to *solve_band_real_rhs_multi()*, *solve_mat_real_rhs_multi()*, *gemm()*

subroutine algebra/**prepare_bordered**(*a1*, *a2*, *a3*, *a4*, *lena1*, *n_boundaries*, *kl*, *ku*, *pivota1*, *pivota4*, *info*)

– Input variables

Parameters

- **a1** (1 + 2 * kl + ku,lena1) *[real,inout]*
- **a2** (lena1,n_boundaries) *[real,inout]*
- **a3** (lena1) *[real,inout]*
- **a4** (n_boundaries,n_boundaries) *[real,inout]*
- **lena1** *[integer,in,]*
- **n_boundaries** *[integer,in,]*
- **pivota1** (lena1) *[integer,out]*
- **pivota4** (n_boundaries) *[integer,out]*
- **info** *[integer,out]*

Options

- **kl** *[integer,in,optional/default=(-1 - ku + shape(a1, 0)) / 2]*
- **ku** *[integer,in,optional/default=-1 - 2 * kl + shape(a1, 0)]*

Called from *prepare()*

Call to *prepare_band()*, *solve_band_real_rhs_multi()*, *prepare_mat()*

subroutine algebra/**gemm**(*m*, *n*, *k*, *alpha*, *a*, *lda*, *b*, *ldb*, *beta*, *c*, *ldc*)

Computes a matrix-matrix product : C <- alpha*A*B+beta*C

– Input variables

Parameters

- **m** *[integer,in]*
- **n** *[integer,in]*

- **k** *[integer,in]*
- **alpha** *[real,in]*
- **a** (lda,*) *[real,in]*
- **lda** *[integer,in,]*
- **b** (ldb,*) *[real,in]*
- **ldb** *[integer,in,]*
- **beta** *[real,in]*
- **c** (ldc,*) *[real,inout]*
- **ldc** *[integer,in,]*

Called from *solve_bordered_real_rhs_multi()*

subroutine algebra/**gemv**(*m*, *n*, *alpha*, *a*, *lda*, *x*, *beta*, *y*)

Computes a matrix-vector product: y <- alpha*A*x+beta

Parameters

- **m** *[integer,in]*
- **n** *[integer,in]*
- **alpha** *[real,in]*
- **a** (lda,*) *[real,in]*
- **lda** *[integer,in,]*
- **x** (*) *[real,in]*
- **beta** *[real,in]*
- **y** (*) *[real,inout]*

Called from *solve_bordered_real_rhs()*, *solve_bordered_complex_rhs()*

10.15 Radial derivatives and integration

10.15.1 radial_derivatives.f90

Description

Radial derivatives functions

Quick access

Variables *get_dcheb*, *get_dr*, *work_1d_real*

Routines *bulk_to_ghost()*, *exch_ghosts()*, *finalize_der_arrays()*, *get_bound_vals()*, *get_dcheb_complex()*, *get_dcheb_real_1d()*, *get_ddcheb()*, *get_dddcheb()*, *get_ddddr_ghost()*, *get_dddr()*, *get_ddr()*, *get_ddr_ghost()*, *get_ddr_rloc()*, *get_dr_complex()*, *get_dr_real_1d()*, *get_dr_rloc()*, *initialize_der_arrays()*

Needed modules

- *constants* (*zero()*, *one()*, *three()*): module containing constants and parameters used in the code.
- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc*: This little module is used to estimate the global memory allocation used in MagIC
- *cosine_transform_odd*: This module contains the built-in type I discrete Cosine Transforms. This implementation is based on Numerical Recipes and FFTPACK. This only works for `n_r_max-1 = 2**a 3**b 5**c`, with a,b,c integers. . . .
- *radial_scheme* (`type_rscheme()`): This is an abstract type that defines the radial scheme used in MagIC
- *logic* (*l_finite_diff()*): Module containing the logicals that control the run
- *parallel_mod*: This module contains the blocking information
- *useful* (*abortrun()*): This module contains several useful routines.

Variables

- radial_der/**get_dcheb** *[public]*
- radial_der/**get_dr** *[public]*
- radial_der/**work** (*,*) *[complex,private/allocatable]*
- radial_der/**work_1d_real** (*) *[real,private/allocatable]*

Subroutines and functions

subroutine radial_der/**initialize_der_arrays**(*n_r_max*, *llm*, *ulm*)

Allocate work arrays to compute derivatives

Parameters

- **n_r_max** *[integer,in]*
- **llm** *[integer,in]*
- **ulm** *[integer,in]*

Called from *magic*

subroutine radial_der/**finalize_der_arrays**()

Deallocate work arrays

Called from *magic*

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*)

Returns Chebyshev coeffitients of first derivative df and second derivative ddf for a function whose cheb-coeff. are given as columns in array f(n_f_max,n_r_max).

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,in]*
- **df** (n_f_max,*n_r_max*) *[complex,out]*
- **n_f_max** *[integer,in,]* :: Max no of functions
- **n_f_start** *[integer,in]* :: No of function to start with
- **n_f_stop** *[integer,in]* :: No of function to stop with
- **n_r_max** *[integer,in,]* :: second dimension of f,df,ddf
- **n_cheb_max** *[integer,in]* :: Number of cheb modes
- **d_fac** *[real,in]* :: factor for interval mapping

subroutine radial_der/**get_dcheb_real_1d**(*f*, *df*, *n_r_max*, *n_cheb_max*, *d_fac*)

Parameters

- **f** (*n_r_max*) *[real,in]*
- **df** (*n_r_max*) *[real,out]*
- **n_r_max** *[integer,in,]* :: Dimension of f,df,ddf
- **n_cheb_max** *[integer,in]* :: Number of cheb modes
- **d_fac** *[real,in]* :: factor for interval mapping

subroutine radial_der/**get_ddcheb**(*f*, *df*, *ddf*, *n_f_max*, *n_f_start*, *n_f_stop*, *n_r_max*, *n_cheb_max*, *d_fac*)

Returns Chebyshev coefficients of first derivative df and second derivative ddf for a function whose cheb-coeff. are given as columns in array f(n_c_tot,n_r_max).

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,in]*
- **df** (n_f_max,*n_r_max*) *[complex,out]*
- **ddf** (n_f_max,*n_r_max*) *[complex,out]*
- **n_f_max** *[integer,in,]* :: First dimension of f,df,ddf
- **n_f_start** *[integer,in]* :: No of column to start with
- **n_f_stop** *[integer,in]* :: No of column to stop with
- **n_r_max** *[integer,in,]* :: second dimension of f,df,ddf
- **n_cheb_max** *[integer,in]* :: Number of cheb modes
- **d_fac** *[real,in]* :: factor for interval mapping

Called from *get_ddr()*

subroutine radial_der/**get_dddcheb**(*f*, *df*, *ddf*, *dddf*, *n_f_max*, *n_f_start*, *n_f_stop*, *n_r_max*, *n_cheb_max*, *d_fac*)

Returns chebychev coeffitiens of first derivative df and second derivative ddf for a function whose cheb-coeff. are given as columns in array f(n_c_tot,n_r_max).

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,in]*
- **df** (n_f_max,*n_r_max*) *[complex,out]*
- **ddf** (n_f_max,*n_r_max*) *[complex,out]*
- **dddf** (n_f_max,*n_r_max*) *[complex,out]*
- **n_f_max** *[integer,in,]* :: First dimension of f,df,ddf
- **n_f_start** *[integer,in]* :: No of column to start with
- **n_f_stop** *[integer,in]* :: No of column to stop with
- **n_r_max** *[integer,in,]* :: second dimension of f,df,ddf
- **n_cheb_max** *[integer,in]* :: Number of cheb modes
- **d_fac** *[real,in]* :: factor for interval mapping

Called from *get_dddr()*

subroutine radial_der/**get_dr_real_1d**(*f*, *df*, *n_r_max*, *r_scheme*)

Parameters

- **f** (*n_r_max*) *[real,in]*
- **df** (*n_r_max*) *[real,out]* :: first derivative of f
- **n_r_max** *[integer,in,]* :: number of radial grid points
- **r_scheme** *[real]*

subroutine radial_der/**get_dr_complex**(*f*, *df*, *n_f_max*, *n_f_start*, *n_f_stop*, *n_r_max*, *r_scheme*[, *nocopy*[, *l_dct_in*]])

Returns first radial derivative df of the input function f. Array f(n_f_max,*) may contain several functions numbered by the first index. The subroutine calculates the derivaties of the functions f(n_f_start,*) to f(n_f_stop) by transforming to a Chebychev representation using n_r_max radial grid points .

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,inout]*
- **df** (n_f_max,*n_r_max*) *[complex,out]* :: first derivative of f
- **n_f_max** *[integer,in,]* :: first dim of f
- **n_f_start** *[integer,in]* :: first function to be treated
- **n_f_stop** *[integer,in]* :: last function to be treated
- **n_r_max** *[integer,in,]* :: number of radial grid points
- **r_scheme** *[real]*

- **nocopy** *[logical,in,]*
- **l_dct_in** *[logical,in,]*

subroutine radial_der/**get_ddr**(*f*, *df*, *ddf*, *n_f_max*, *n_f_start*, *n_f_stop*, *n_r_max*, *r_scheme*[, *l_dct_in*])

Returns first radial derivative df and second radial derivative ddf of the input function f. Array f(n_f_max,*) may contain several functions numbered by the first index. The subroutine calculates the derivatives of the functions f(n_f_start,*) to f(n_f_stop) by transforming to a Chebychev representation using n_r_max radial grid points.

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,in]*
- **df** (n_f_max,*n_r_max*) *[complex,out]* :: first derivative of f
- **ddf** (n_f_max,*n_r_max*) *[complex,out]* :: second derivative of f
- **n_f_max** *[integer,in,]* :: first dim of f
- **n_f_start** *[integer,in]* :: first function to be treated
- **n_f_stop** *[integer,in]* :: last function to be treated
- **n_r_max** *[integer,in,]* :: number of radial grid points
- **r_scheme** *[real]*
- **l_dct_in** *[logical,in,]*

Called from *get_mag_rhs_imp()*, *get_phase_rhs_imp()*, *get_entropy_rhs_imp()*, *get_pol_rhs_imp()*, *assemble_pol()*, *assemble_single()*, *get_single_rhs_imp()*, *get_comp_rhs_imp()*, *get_tor_rhs_imp()*

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*, *r_scheme*[, *l_dct_in*])

Returns first radial derivative df, the second radial deriv. ddf, and the third radial derivative dddf of the input function f. Array f(n_f_max,*) may contain several functions numbered by the first index. The subroutine calculates the derivaties of the functions f(n_f_start,*) to f(n_f_stop) by transforming to a Chebychev representation using n_r_max radial grid points.

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,in]*
- **df** (n_f_max,*n_r_max*) *[complex,out]* :: first derivative of f
- **ddf** (n_f_max,*n_r_max*) *[complex,out]* :: second derivative of f
- **dddf** (n_f_max,*n_r_max*) *[complex,out]* :: third derivative of f
- **n_f_max** *[integer,in,]* :: first dim of f
- **n_f_start** *[integer,in]* :: first function to be treated
- **n_f_stop** *[integer,in]* :: last function to be treated
- **n_r_max** *[integer,in,]* :: number of radial grid points
- **r_scheme** *[real]*

- **l_dct_in** *[logical,in,]*

Called from *get_pol_rhs_imp()*, *get_single_rhs_imp()*

Call to *get_dddcheb()*

subroutine radial_der/**get_dr_rloc**(*f_rloc*, *df_rloc*, *lm_max*, *nrstart*, *nrstop*, *n_r_max*, *r_scheme*)

Purpose of this subroutine is to take the first radial derivative of an input complex array distributed over radius. This can only be used with finite differences.

Parameters

- **f_rloc** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,in]*
- **df_rloc** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **lm_max** *[integer,in,]*
- **n_r_max** *[integer,in]*
- **r_scheme** *[real]*

Options

- **nrstart** *[integer,in,optional/default=(-1 - nrstop + shape(f_rloc, 1)) / (-1)]*
- **nrstop** *[integer,in,optional/default=-1 + nrstart + shape(f_rloc, 1)]*

Called from *dtvrms()*, *dtbrms()*, *transp_lmloc_to_rloc()*, *transp_rloc_to_lmloc_io()*, *finish_exp_mag_rdist()*, *finish_exp_entropy_rdist()*, *finish_exp_pol_rdist()*, *get_pol_rhs_imp_ghost()*, *finish_exp_smat_rdist()*, *finish_exp_comp_rdist()*

Call to *abortrun()*, *get_openmp_blocks()*, *exch_ghosts()*, *get_bound_vals()*

subroutine radial_der/**get_ddr_rloc**(*f_rloc*, *df_rloc*, *ddf_rloc*, *lm_max*, *nrstart*, *nrstop*, *n_r_max*, *r_scheme*)

Purpose of this subroutine is to take the first and second radial derivatives of an input complex array distributed over radius. This can only be used with finite differences.

Parameters

- **f_rloc** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,in]*
- **df_rloc** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **ddf_rloc** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **lm_max** *[integer,in,]*
- **n_r_max** *[integer,in]*
- **r_scheme** *[real]*

Options

- **nrstart** *[integer,in,optional/default=(-1 - nrstop + shape(f_rloc, 1)) / (-1)]*
- **nrstop** *[integer,in,optional/default=-1 + nrstart + shape(f_rloc, 1)]*

Called from *transp_lmloc_to_rloc()*

Call to *abortrun()*, *get_openmp_blocks()*, *exch_ghosts()*, *get_bound_vals()*

subroutine radial_der/**get_ddr_ghost**(*f_rloc*, *df_rloc*, *ddf_rloc*, *lm_max*, *start_lm*, *stop_lm*, *nrstart*, *nrstop*, *r_scheme*)

Purpose of this subroutine is to take the first and second radial derivatives of an input complex array distributed over radius that has the ghost zones properly filled.

Parameters

- **f_rloc** (*lm_max*,3 - *nrstart* + *nrstop*) *[complex,in]*
- **df_rloc** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **ddf_rloc** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **lm_max** *[integer,in,]*
- **start_lm** *[integer,in]*
- **stop_lm** *[integer,in]*
- **r_scheme** *[real]*

Options

- **nrstart** *[integer,in,optional/default=(-3 - nrstop + shape(f_rloc, 1)) / (-1)]*
- **nrstop** *[integer,in,optional/default=-3 + nrstart + shape(f_rloc, 1)]*

Called from *get_mag_rhs_imp_ghost()*, *get_phase_rhs_imp_ghost()*, *get_entropy_rhs_imp_ghost()*, *get_comp_rhs_imp_ghost()*, *get_tor_rhs_imp_ghost()*

Call to *abortrun()*

subroutine radial_der/**get_ddddr_ghost**(*f_rloc*, *df_rloc*, *ddf_rloc*, *dddf_rloc*, *ddddf_rloc*, *lm_max*, *start_lm*, *stop_lm*, *nrstart*, *nrstop*, *r_scheme*)

Purpose of this subroutine is to take the first and second radial derivatives of an input complex array distributed over radius that has the ghost zones properly filled.

Parameters

- **f_rloc** (*lm_max*,5 - *nrstart* + *nrstop*) *[complex,in]*
- **df_rloc** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **ddf_rloc** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **dddf_rloc** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **ddddf_rloc** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,out]*
- **lm_max** *[integer,in,]*
- **start_lm** *[integer,in]*
- **stop_lm** *[integer,in]*
- **r_scheme** *[real]*

Options

- **nrstart** *[integer,in,optional/default=(-5 - nrstop + shape(f_rloc, 1)) / (-1)]*
- **nrstop** *[integer,in,optional/default=-5 + nrstart + shape(f_rloc, 1)]*

Called from *get_pol_rhs_imp_ghost()*

Call to *abortrun()*

subroutine radial_der/**exch_ghosts**(*f*, *lm_max*, *nrstart*, *nrstop*, *nghosts*)

Parameters

- **f** (*lm_max*,1 + 2 * nghosts - *nrstart* + *nrstop*) *[complex,inout]*
- **lm_max** *[integer,in,]*

Options

- **nrstart** *[integer,in,optional/default=(-1 - 2 * nghosts - nrstop + shape(f, 1)) / (-1)]*
- **nrstop** *[integer,in,optional/default=-1 - 2 * nghosts + nrstart + shape(f, 1)]*
- **nghosts** *[integer,in,optional/default=(-1 + nrstart - nrstop + shape(f, 1)) / 2]*

Called from *get_dr_rloc()*, *get_ddr_rloc()*, *getstartfields()*, *start_from_another_scheme()*, *assemble_mag_rloc()*, *assemble_phase_rloc()*, *assemble_entropy_rloc()*, *assemble_pol_rloc()*, *assemble_comp_rloc()*, *assemble_tor_rloc()*

subroutine radial_der/**get_bound_vals**(*fbot*, *ftop*, *lm_max*, *nrstart*, *nrstop*, *n_r_max*, *nbounds*)

Parameters

- **fbot** (*lm_max*,nbounds) *[complex,inout]*
- **ftop** (*lm_max*,nbounds) *[complex,inout]*
- **lm_max** *[integer,in,]*
- **nrstart** *[integer,in]*
- **nrstop** *[integer,in]*
- **n_r_max** *[integer,in]*
- **nbounds** *[integer,in,]*

Called from *get_dr_rloc()*, *get_ddr_rloc()*

subroutine radial_der/**bulk_to_ghost**(*x*, *x_g*, *ng*, *nrstart*, *nrstop*, *lm_max*, *start_lm*, *stop_lm*)

This subroutine is used to copy an array that is defined from nRstart to nRstop to an array that is defined from nRstart-1 to nRstop+1

Parameters

- **x** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,in]*
- **x_g** (*lm_max*,1 + 2 * ng - *nrstart* + *nrstop*) *[complex,out]*
- **ng** *[integer,in]* :: Number of ghost zones
- **lm_max** *[integer,in,]*
- **start_lm** *[integer,in]*
- **stop_lm** *[integer,in]*

Options

- **nrstart** *[integer,in,optional/default=(-1 - nrstop + shape(x, 1)) / (-1)]*
- **nrstop** *[integer,in,optional/default=-1 + nrstart + shape(x, 1)]*

Called from *getstartfields()*, *start_from_another_scheme()*, *assemble_mag_rloc()*, *assemble_phase_rloc()*, *assemble_entropy_rloc()*, *assemble_pol_rloc()*, *assemble_comp_rloc()*, *assemble_tor_rloc()*

10.15.2 radial_derivatives_even.f90

Quick access

Routines *get_dcheb_even()*, *get_ddcheb_even()*, *get_ddr_even()*, *get_ddrns_even()*, *get_drns_even()*

Needed modules

- *constants* (*zero()*): module containing constants and parameters used in the code.
- *precision_mod*: This module controls the precision used in MagIC
- *cosine_transform_odd*: This module contains the built-in type I discrete Cosine Transforms. This implementation is based on Numerical Recipes and FFTPACK. This only works for `n_r_max-1 = 2**a 3**b 5**c`, with a,b,c integers....
- *cosine_transform_even*

Variables

Subroutines and functions

subroutine radial_der_even/**get_ddr_even**(*f*, *df*, *ddf*, *n_f_max*, *n_f_start*, *n_f_stop*, *n_r_max*, *n_cheb_max*, *dr_fac*, *work1*, *work2*, *chebt_odd*, *chebt_even*)

Returns first rarial derivative df and second radial derivative ddf of the input function f. Array f(n_f_max,*) may contain several functions numbered by the first index. The subroutine calculates the derivaties of the functions f(n_f_start,*) to f(n_f_stop) by transforming to a Chebychev representation using n_r_max radial grid points. The cheb transforms have to be initialized by calling init_costf1 and init_costf2.

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,in]*
- **df** (n_f_max,*n_r_max*) *[complex,out]* :: first derivative of f
- **ddf** (n_f_max,*n_r_max*) *[complex,out]* :: second derivative of f
- **n_f_max** *[integer,in,]* :: first dim of f
- **n_f_start** *[integer,in]* :: first function to be treated
- **n_f_stop** *[integer,in]* :: last function to be treated
- **n_r_max** *[integer,in,]* :: number of radial grid points
- **n_cheb_max** *[integer,in]* :: number of cheb modes

- **dr_fac** *[real,in]* :: mapping factor
- **work1** (n_f_max,*n_r_max*) *[complex,out]* :: work array needed for costf
- **work2** (n_f_max,*n_r_max*) *[complex,out]* :: work array needed for costf
- **chebt_odd** *[costf_odd_t,in]*
- **chebt_even** *[costf_even_t,in]*

Called from *get_mag_ic_rhs_imp()*

Call to *get_ddcheb_even()*

subroutine radial_der_even/**get_drns_even**(*f*, *df*, *n_f_max*, *n_f_start*, *n_f_stop*, *n_r_max*, *n_cheb_max*, *dr_fac*, *work1*, *chebt_odd*, *chebt_even*)

Returns first rarial derivative df and second radial derivative ddf of the input function f. Array f(n_f_max,*) may contain several functions numbered by the first index. The subroutine calculates the derivaties of the functions f(n_f_start,*) to f(n_f_stop) by transforming to a Chebychev representation using n_r_max radial grid points. The cheb transforms have to be initialized by calling init_costf1 and init_costf2.

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,inout]*
- **df** (n_f_max,*n_r_max*) *[complex,out]* :: first derivative of f
- **n_f_max** *[integer,in,]* :: first dim of f
- **n_f_start** *[integer,in]* :: first function to be treated
- **n_f_stop** *[integer,in]* :: last function to be treated
- **n_r_max** *[integer,in,]* :: number of radial grid points
- **n_cheb_max** *[integer,in]* :: number of cheb modes
- **dr_fac** *[real,in]* :: mapping factor
- **work1** (n_f_max,*n_r_max*) *[complex,out]* :: work array needed for costf
- **chebt_odd** *[costf_odd_t,in]*
- **chebt_even** *[costf_even_t,in]*

Called from *fields_average()*, *write_dtb_frame()*

Call to *get_dcheb_even()*

subroutine radial_der_even/**get_ddrns_even**(*f*, *df*, *ddf*, *n_f_max*, *n_f_start*, *n_f_stop*, *n_r_max*, *n_cheb_max*, *dr_fac*, *work1*, *chebt_odd*, *chebt_even*)

Returns first rarial derivative df and second radial derivative ddf of the input function f. Array f(n_f_max,*) may contain several functions numbered by the first index. The subroutine calculates the derivaties of the functions f(n_f_start,*) to f(n_f_stop) by transforming to a Chebychev representation using n_r_max radial grid points. The cheb transforms have to be initialized by calling init_costf1 and init_costf2.

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,inout]*

- **df** (n_f_max,*n_r_max*) *[complex,out]* :: first derivative of f
- **ddf** (n_f_max,*n_r_max*) *[complex,out]* :: second derivative of f
- **n_f_max** *[integer,in,]* :: first dim of f
- **n_f_start** *[integer,in]* :: first function to be treated
- **n_f_stop** *[integer,in]* :: last function to be treated
- **n_r_max** *[integer,in,]* :: number of radial grid points
- **n_cheb_max** *[integer,in]* :: number of cheb modes
- **dr_fac** *[real,in]* :: mapping factor
- **work1** (n_f_max,*n_r_max*) *[complex,out]* :: work array needed for costf
- **chebt_odd** *[costf_odd_t,in]*
- **chebt_even** *[costf_even_t,in]*

Called from *fields_average()*

Call to *get_ddcheb_even()*

subroutine radial_der_even/**get_dcheb_even**(*f*, *df*, *n_f_max*, *n_f_start*, *n_f_stop*, *n_r_max*, *n_cheb_max*, *d_fac*)

Returns Chebyshev coefficients of first derivative df and second derivative ddf for a function whose cheb-coeff. are given as columns in array f(n_f_max,n_r_max).

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,in]*
- **df** (n_f_max,*n_r_max*) *[complex,out]*
- **n_f_max** *[integer,in,]* :: First dimension of f,df
- **n_f_start** *[integer,in]* :: No of function to start with
- **n_f_stop** *[integer,in]* :: No of function to stop with
- **n_r_max** *[integer,in,]* :: second dimension of f,df
- **n_cheb_max** *[integer,in]* :: Number of cheb modes
- **d_fac** *[real,in]* :: factor for interval mapping

Called from *get_drns_even()*

subroutine radial_der_even/**get_ddcheb_even**(*f*, *df*, *ddf*, *n_f_max*, *n_f_start*, *n_f_stop*, *n_r_max*, *n_cheb_max*, *d_fac*)

Returns Chebyshev coefficients of first derivative df and second derivative ddf for a function whose cheb-coeff. are given as columns in array f(n_f_max,n_r_max).

Parameters

- **f** (n_f_max,*n_r_max*) *[complex,in]*
- **df** (n_f_max,*n_r_max*) *[complex,out]*
- **ddf** (n_f_max,*n_r_max*) *[complex,out]*

- **n_f_max** *[integer,in,]* :: First dimension of f,df,ddf
- **n_f_start** *[integer,in]* :: No of function to start with
- **n_f_stop** *[integer,in]* :: No of function to stop with
- **n_r_max** *[integer,in,]* :: second dimension of f,df,ddf
- **n_cheb_max** *[integer,in]* :: Number of cheb modes
- **d_fac** *[real,in]* :: factor for interval mapping

Called from *get_ddr_even()*, *get_ddrns_even()*

10.15.3 integration.f90

Description

Radial integration functions

Quick access

Routines *cylmean_itc()*, *cylmean_otc()*, *rint_r()*, *rintic()*, *simps()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *constants* (*half()*, *one()*, *two()*, *pi()*): module containing constants and parameters used in the code.
- *radial_scheme* (`type_rscheme()`): This is an abstract type that defines the radial scheme used in MagIC
- *cosine_transform_odd*: This module contains the built-in type I discrete Cosine Transforms. This implementation is based on Numerical Recipes and FFTPACK. This only works for `n_r_max-1 = 2**a 3**b 5**c`, with a,b,c integers....

Variables

Subroutines and functions

function integration/**rintic**(*f*, *nrmax*, *drfac*, *chebt*)

This function performs the radial integral over a function f that is given on the appropriate nRmax radial Chebychev grid points.

Parameters

- **f** (nrmax) *[real,inout]* :: This is zero order contribution
- **nrmax** *[integer,in,]* :: Only even chebs for IC
- **drfac** *[real,in]*
- **chebt** *[costf_odd_t,in]*

Return **rintic** *[real]*

Called from *get_e_mag()*, *get_power()*, *spectrum()*

function integration/**rint_r**(*f*, *r*, *r_scheme*)

Same as function rInt but for a radial dependent mapping function dr_fac2.

Parameters

- **f** (*) *[real,in]* :: Input function
- **r** (*) *[real,in]* :: Radius
- **r_scheme** *[real]* :: Radial scheme (FD or Cheb)

Return **rint** *[real]*

Called from *get_force()*, *dtbrms()*, *get_poltorrms()*, *getdlm()*, *get_e_kin()*, *get_u_square()*, *get_e_mag()*, *outhelicity()*, *outheat()*, *outphase()*, *outperppar()*, *get_angular_moment()*, *get_angular_moment_rloc()*, *output()*, *get_power()*, *precalc()*, *spectrum()*, *spectrum_temp()*, *get_amplitude()*, *updatewp()*

Call to *simps()*

function integration/**simps**(*f*, *r*)

Simpson's method to integrate a function

Parameters

- **f** (*) *[real,in]* :: Input function
- **r** (*) *[real,in]* :: Radius

Return **rint** *[real]*

Called from *rint_r()*, *initialize_geos()*, *outgeos()*, *initialize_outto_mod()*, *outto()*

subroutine integration/**cylmean_otc**(*a*, *v*, *n_s_max*, *n_s_otc*, *r*, *s*, *theta*[, *zdensin*])

This routine computes a z-averaging using Simpsons's rule outside T.C.

Parameters

- **a** (,) *[real,in]*
- **v** (*n_s_max*) *[real,out]*
- **n_s_max** *[integer,in,]*
- **n_s_otc** *[integer,in]*
- **r** (*) *[real,in]* :: Spherical radius
- **s** (*n_s_max*) *[real,in]* :: Cylindrical radius
- **theta** (*) *[real,in]* :: Colatitude
- **zdensin** *[real,in,]*

Called from *outgeos()*, *outomega()*, *cylmean()*

subroutine integration/**cylmean_itc**(*a*, *vn*, *vs*, *n_s_max*, *n_s_otc*, *r*, *s*, *theta*[, *zdensin*])

This routine computes a z-averaging using Simpsons's rule inside T.C.

Parameters

- **a** (,) *[real,in]*
- **vn** (*n_s_max*) *[real,out]*
- **vs** (*n_s_max*) *[real,out]*
- **n_s_max** *[integer,in,]*
- **n_s_otc** *[integer,in]*
- **r** (*) *[real,in]* :: Spherical radius
- **s** (*n_s_max*) *[real,in]* :: Cylindrical radius
- **theta** (*) *[real,in]* :: Colatitude
- **zdensin** *[real,in,]*

Called from *outomega()*, *cylmean()*

10.16 Blocking and LM mapping

10.16.1 blocking.f90

Description

Module containing blocking information

Quick access

Variables *l2lmas*, *llm*, *llmmag*, *lm2*, *lm22l*, *lm22lm*, *lm22m*, *lm2l*, *lm2lma*, *lm2lmp*, *lm2lms*, *lm2m*, *lm2mc*, *lm_balance*, *lmp2*, *lmp2l*, *lmp2lm*, *lmp2lmpa*, *lmp2lmps*, *lo_map*, *lo_sub_map*, *nlmbs2*, *sizelmb2*, *sn_sub_map*, *st_map*, *st_sub_map*, *ulm*, *ulmmag*

Routines *finalize_blocking()*, *get_lorder_lm_blocking()*, *get_snake_lm_blocking()*, *get_standard_lm_blocking()*, *get_subblocks()*, *initialize_blocking()*

Needed modules

- iso_fortran_env (output_unit())
- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc* (*memwrite()*, *bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *parallel_mod* (*nthreads()*, rank(), *n_procs()*, *rank_with_l1m0()*, load(), *getblocks()*): This module contains the blocking information
- *truncation* (*lmp_max()*, *lm_max()*, *l_max()*, *n_theta_max()*, *minc()*, *n_r_max()*, *m_max()*, *l_axi()*): This module defines the grid points and the truncation
- *logic* (*l_save_out()*, *l_finite_diff()*, *l_mag()*): Module containing the logicals that control the run
- *output_data* (*n_log_file()*, *log_file()*): This module contains the parameters for output control

- *lmmapping* (mappings(), *allocate_mappings()*, *deallocate_mappings()*, *allocate_subblocks_mappings()*, *deallocate_subblocks_mappings()*, subblocks_mappings())
- *useful* (*logwrite()*, *abortrun()*): This module contains several useful routines.
- *constants* (*one()*): module containing constants and parameters used in the code.

Variables

- blocking/**l2lmas** (*) *[integer,pointer/public]*
- blocking/**llm** *[integer,public]*
- blocking/**llmmag** *[integer,public]*
- blocking/**lm2** (*,*) *[integer,pointer/public]*
- blocking/**lm22l** (*,*,*) *[integer,pointer/public]*
- blocking/**lm22lm** (*,*,*) *[integer,pointer/public]*
- blocking/**lm22m** (*,*,*) *[integer,pointer/public]*
- blocking/**lm2l** (*) *[integer,pointer/public]*
- blocking/**lm2lma** (*) *[integer,pointer/public]*
- blocking/**lm2lmp** (*) *[integer,pointer/public]*
- blocking/**lm2lms** (*) *[integer,pointer/public]*
- blocking/**lm2m** (*) *[integer,pointer/public]*
- blocking/**lm2mc** (*) *[integer,pointer/public]*
- blocking/**lm_balance** (*) *[load,allocatable/public]*
- blocking/**lmp2** (*,*) *[integer,pointer/public]*
- blocking/**lmp2l** (*) *[integer,pointer/public]*
- blocking/**lmp2lm** (*) *[integer,pointer/public]*
- blocking/**lmp2lmpa** (*) *[integer,pointer/public]*
- blocking/**lmp2lmps** (*) *[integer,pointer/public]*
- blocking/**lo_map** *[mappings,target/public]*
- blocking/**lo_sub_map** *[subblocks_mappings,target/public]*
- blocking/**nlmbs2** (*) *[integer,pointer/public]*
- blocking/**sizelmb2** (*,*) *[integer,pointer/public]*
- blocking/**sn_sub_map** *[subblocks_mappings,target/public]*
- blocking/**st_map** *[mappings,target/public]*
- blocking/**st_sub_map** *[subblocks_mappings,target/public]*
- blocking/**ulm** *[integer,public]*
- blocking/**ulmmag** *[integer,public]*

Subroutines and functions

subroutine blocking/**initialize_blocking**()

Called from *magic*

Call to *allocate_mappings()*, *abortrun()*, *getblocks()*, *get_standard_lm_blocking()*, *get_snake_lm_blocking()*, *get_lorder_lm_blocking()*, *allocate_subblocks_mappings()*, *get_subblocks()*, *memwrite()*

subroutine blocking/**finalize_blocking**()

Called from *magic*

Call to *deallocate_mappings()*, *deallocate_subblocks_mappings()*

subroutine blocking/**get_subblocks**(*map*, *sub_map*)

Parameters

- **map** *[mappings,in]*
- **sub_map** *[subblocks_mappings,inout]*

Called from *initialize_blocking()*

Call to *abortrun()*

subroutine blocking/**get_standard_lm_blocking**(*map*, *minc*)

Parameters

- **map** *[mappings,inout]* :: Extra l for lmP
- **minc** *[integer,in]*

Called from *initialize_blocking()*

Call to *abortrun()*

subroutine blocking/**get_lorder_lm_blocking**(*map*, *minc*)

Parameters

- **map** *[mappings,inout]* :: Extra l for lmP
- **minc** *[integer,in]*

Called from *initialize_blocking()*

Call to *abortrun()*

subroutine blocking/**get_snake_lm_blocking**(*map*, *minc*, *lm_balance*)

Parameters

- **map** *[mappings,inout]* :: Extra l for lmP
- **minc** *[integer,in]*
- **lm_balance** (*n_procs*) *[load,inout]*

Called from *initialize_blocking()*

Call to *abortrun()*

10.16.2 LMmapping.f90

Quick access

Routines *allocate_mappings()*, *allocate_subblocks_mappings()*, *deallocate_mappings()*, *deallocate_subblocks_mappings()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *truncation* (*l_axi()*): This module defines the grid points and the truncation
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *parallel_mod* (load()): This module contains the blocking information

Types

- **type** lmmapping/**unknown_type**

  Type fields

  - **% l2lmas** (*) *[integer,allocatable]*
  - **% l_max** *[integer]*
  - **% lm2** (,) *[integer,allocatable]*
  - **% lm2l** (*) *[integer,allocatable]*
  - **% lm2lma** (*) *[integer,allocatable]*
  - **% lm2lmp** (*) *[integer,allocatable]*
  - **% lm2lms** (*) *[integer,allocatable]*
  - **% lm2m** (*) *[integer,allocatable]*
  - **% lm2mc** (*) *[integer,allocatable]*
  - **% lm_max** *[integer]*
  - **% lmp2** (,) *[integer,allocatable]*
  - **% lmp2l** (*) *[integer,allocatable]*
  - **% lmp2lm** (*) *[integer,allocatable]*
  - **% lmp2lmpa** (*) *[integer,allocatable]*
  - **% lmp2lmps** (*) *[integer,allocatable]*
  - **% lmp2m** (*) *[integer,allocatable]*
  - **% lmp_max** *[integer]*
  - **% m_max** *[integer]*

- **type** lmmapping/**unknown_type**

  Type fields

  - **% l_max** *[integer]*
  - **% lm22l** (,,*) *[integer,allocatable]*
  - **% lm22lm** (,,*) *[integer,allocatable]*
  - **% lm22m** (,,*) *[integer,allocatable]*
  - **% m_max** *[integer]*
  - **% nlmbs** *[integer]*
  - **% nlmbs2** (*) *[integer,allocatable]*
  - **% sizelmb2** (,) *[integer,allocatable]*
  - **% sizelmb2max** *[integer]*

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/**deallocate_mappings**(*self*)

Parameters **self** *[mappings]*

Called from *finalize_blocking()*

subroutine lmmapping/**allocate_subblocks_mappings**(*self*, *map*, *nlmbs*, *l_max*, *lm_balance*)

Parameters

- **self** *[subblocks_mappings]*
- **map** *[mappings,in]*
- **nlmbs** *[integer,in,]*
- **l_max** *[integer,in]*
- **lm_balance** (nlmbs) *[load,in]*

Called from *initialize_blocking()*

subroutine lmmapping/**deallocate_subblocks_mappings**(*self*)

Parameters **self** *[subblocks_mappings]*

Called from *finalize_blocking()*

10.17 IO: time series, radial profiles and spectra

10.17.1 output.f90

Description

This module handles the calls to the different output routines.

Quick access

Variables *cmb_file*, *cmbmov_file*, *dipcmbmean*, *dipmean*, *dlbmean*, *dlvcmean*, *dlvmean*, *dmbmean*, *dmvmean*, *dpvmean*, *dt_cmb_file*, *dte_file*, *dteint*, *dzvmean*, *e_kin_pmean*, *e_kin_tmean*, *e_mag_pmean*, *e_mag_tmean*, *elcmbmean*, *elmean*, *etot*, *etotold*, *geosamean*, *geosmean*, *geosmmean*, *geosnapmean*, *geoszmean*, *lbdissmean*, *lvdissmean*, *n_cmb_file*, *n_cmb_setsmov*, *n_cmbmov_file*, *n_dt_cmb_file*, *n_dt_cmb_sets*, *n_dte_file*, *n_e_sets*, *n_par_file*, *n_spec*, *nlogs*, *npotsets*, *nrms_sets*, *par_file*, *rela*, *relm*, *relna*, *relz*, *rmmean*, *rolmean*, *timenormlog*, *timenormrms*, *timepassedlog*, *timepassedrms*

Routines *finalize_output()*, *initialize_output()*, *output()*

Needed modules

- iso_fortran_env (output_unit())
- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod*: This module contains the blocking information
- *truncation* (*n_r_max()*, *n_r_ic_max()*, *minc()*, *l_max()*, *l_maxmag()*, *n_r_maxmag()*, *lm_max()*): This module defines the grid points and the truncation
- *radial_functions* (*or1()*, *or2()*, *r()*, rscheme_oc(), *r_cmb()*, *r_icb()*, *orho1()*, *sigma()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *radial_data* (*nrstart()*, *nrstop()*, *nrstartmag()*, *nrstopmag()*, *n_r_cmb()*, *n_r_icb()*): This module defines the MPI decomposition in the radial direction.
- *physical_parameters* (*opm()*, *ek()*, *ktopv()*, *prmag()*, *nvarcond()*, *lffac()*, *ekscaled()*): Module containing the physical parameters
- *num_param* (*tscale()*, *escale()*): Module containing numerical and control parameters
- *blocking* (*st_map()*, *lm2()*, *lo_map()*, *llm()*, *ulm()*, *llmmag()*, *ulmmag()*): Module containing blocking information
- *horizontal_data* (*hdif_b()*, *dpl0eq()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *logic* (*l_average()*, *l_mag()*, *l_power()*, *l_anel()*, *l_mag_lf()*, *lverbose()*, *l_dtb()*, *l_rms()*, *l_r_field()*, *l_r_fieldt()*, *l_r_fieldxi()*, *l_sric()*, *l_cond_ic()*, *l_rmagspec()*, *l_movie_ic()*, *l_store_frame()*, *l_cmb_field()*, *l_dt_cmb_field()*, *l_save_out()*, *l_non_rot()*, *l_perppar()*, *l_energy_modes()*, *l_heat()*, *l_hel()*, *l_par()*, *l_chemical_conv()*, *l_movie()*, *l_full_sphere()*, *l_spec_avg()*, *l_phase_field()*): Module containing the logicals that control the run

- *fields* (*omega_ic()*, *omega_ma()*, *b_ic()*, *db_ic()*, *ddb_ic()*, *aj_ic()*, *dj_ic()*, *ddj_ic()*, *w_lmloc()*, *dw_lmloc()*, *ddw_lmloc()*, *p_lmloc()*, *xi_lmloc()*, *s_lmloc()*, *ds_lmloc()*, *z_lmloc()*, *dz_lmloc()*, *b_lmloc()*, *db_lmloc()*, *ddb_lmloc()*, *aj_lmloc()*, *dj_lmloc()*, *ddj_lmloc()*, *b_ic_lmloc()*, *db_ic_lmloc()*, *ddb_ic_lmloc()*, *aj_ic_lmloc()*, *dj_ic_lmloc()*, *ddj_ic_lmloc()*, *dp_lmloc()*, *dxi_lmloc()*, *w_rloc()*, *z_rloc()*, *p_rloc()*, *s_rloc()*, *xi_rloc()*, *b_rloc()*, *aj_rloc()*, *bicb()*, *phi_rloc()*, *phi_lmloc()*): This module contains all the fields used in MagIC in the hybrid (LM,r) space as well as their radial derivatives. It defines both the LM-distributed arrays and the R-distributed arrays....
- *fieldslast* (*dwdt()*, *dzdt()*, *dpdt()*, *dsdt()*, *dbdt()*, *djdt()*, *dbdt_ic()*, *dphidt()*, *djdt_ic()*, *dxidt()*, *domega_ic_dt()*, *domega_ma_dt()*, *lorentz_torque_ma_dt()*, *lorentz_torque_ic_dt()*): This module contains all the work arrays of the previous time-steps needed to time advance the code. They are needed in the time-stepping scheme....
- *kinetic_energy* (*get_e_kin()*, *get_u_square()*)
- *magnetic_energy* (*get_e_mag()*): This module handles the computation and the writing of the diagnostic files related to magnetic energy: e_mag_oc.TAG, e_mag_ic.TAG, dipole.TAG, eMagR.TAG
- *fields_average_mod* (*fields_average()*): This module is used when one wants to store time-averaged quantities
- *spectra* (*spectrum()*, *spectrum_temp()*, *get_amplitude()*)
- *outto_mod* (*outto()*): This module handles the writing of TO-related outputs: zonal force balance and z-integrated terms. This is a re-implementation of the spectral method used up to MagIC 5.10, which formerly relies on calculation of Plm on the cylindrical grid....
- *output_data* (*tag()*, *l_max_cmb()*, *n_coeff_r()*, *l_max_r()*, *n_coeff_r_max()*, *n_r_array()*, *n_r_step()*, *n_log_file()*, *log_file()*): This module contains the parameters for output control
- *constants* (*vol_oc()*, *vol_ic()*, *mass()*, *surf_cmb()*, *two()*, *three()*): module containing constants and parameters used in the code.
- *outmisc_mod* (*outhelicity()*, *outheat()*, *outphase()*): This module contains several subroutines that can compute and store various informations: helicity, heat transfer.
- *geos* (*outgeos()*, *outomega()*): This module is used to compute z-integrated diagnostics such as the degree of geostrophy or the separation of energies between inside and outside the tangent cylinder. This makes use of a local Simpson's method. This also
- *outrot* (*write_rot()*): This module handles the writing of several diagnostic files related to the rotation: angular momentum (AM.TAG), drift (drift.TAG), inner core and mantle rotations....
- *integration* (*rint_r()*): Radial integration functions
- *outpar_mod* (*outpar()*, *outperppar()*): This module is used to compute several time-averaged radial profiles: fluxes, boundary layers, etc.
- *graphout_mod* (*graphout_ic()*): This module contains the subroutines that store the 3-D graphic files.
- *power* (*get_power()*): This module handles the writing of the power budget
- *communications* (*gather_all_from_lo_to_rank0()*, *gt_oc()*, *gt_ic()*, *gather_from_lo_to_rank0()*): This module contains the different MPI communicators used in MagIC.
- *out_coeff* (*write_pot_mpi()*): This module contains the subroutines that calculate the Bcmb files, the [B|V|T]_coeff_r files and the [B|V|T]_lmr files
- *getdlm_mod* (*getdlm()*): This module is used to calculate the lengthscales
- *movie_data* (*movie_gather_frames_to_rank0()*)

- *dtb_mod* (*get_dtblmfinish()*): This module contains magnetic field stretching and advection terms plus a separate omega-effect. It is used for movie output....
- *out_movie* (*write_movie_frame()*)
- *out_movie_ic* (*store_movie_frame_ic()*)
- *rms* (*zerorms()*, *dtvrms()*, *dtbrms()*): This module contains the calculation of the RMS force balance and induction terms.
- *useful* (*logwrite()*): This module contains several useful routines.
- *radial_spectra*
- *time_schemes* (`type_tscheme()`): This module defines an abstract class type_tscheme which is employed for the time advance of the code.
- *storecheckpoints*: This module contains several subroutines that can be used to store the checkpoint_#.tag files

Variables

- output_mod/**cmb_file** *[character,private]*
- output_mod/**cmbmov_file** *[character,private]*
- output_mod/**dipcmbmean** *[real,private]*
- output_mod/**dipmean** *[real,private]*
- output_mod/**dlbmean** *[real,private]*
- output_mod/**dlvcmean** *[real,private]*
- output_mod/**dlvmean** *[real,private]*
- output_mod/**dmbmean** *[real,private]*
- output_mod/**dmvmean** *[real,private]*
- output_mod/**dpvmean** *[real,private]*
- output_mod/**dt_cmb_file** *[character,private]*
- output_mod/**dte_file** *[character,private]*
- output_mod/**dteint** *[real,private]*
- output_mod/**dzvmean** *[real,private]*
- output_mod/**e_kin_pmean** *[real,private]*
- output_mod/**e_kin_tmean** *[real,private]*
- output_mod/**e_mag_pmean** *[real,private]*
- output_mod/**e_mag_tmean** *[real,private]*
- output_mod/**elcmbmean** *[real,private]*
- output_mod/**elmean** *[real,private]*
- output_mod/**etot** *[real,private]*
- output_mod/**etotold** *[real,private]*
- output_mod/**geosamean** *[real,private]*

- output_mod/**geosmean** *[real,private]*
- output_mod/**geosmmean** *[real,private]*
- output_mod/**geosnapmean** *[real,private]*
- output_mod/**geoszmean** *[real,private]*
- output_mod/**lbdissmean** *[real,private]*
- output_mod/**lvdissmean** *[real,private]*
- output_mod/**n_cmb_file** *[integer,private]*
- output_mod/**n_cmb_setsmov** *[integer,private]*
- output_mod/**n_cmbmov_file** *[integer,private]*
- output_mod/**n_dt_cmb_file** *[integer,private]*
- output_mod/**n_dt_cmb_sets** *[integer,private]*
- output_mod/**n_dte_file** *[integer,private]*
- output_mod/**n_e_sets** *[integer,private]*
- output_mod/**n_par_file** *[integer,private]*
- output_mod/**n_spec** *[integer,private]*
- output_mod/**nlogs** *[integer,private]*
- output_mod/**npotsets** *[integer,private]*
- output_mod/**nrms_sets** *[integer,private]*
- output_mod/**par_file** *[character,private]*
- output_mod/**rela** *[real,private]*
- output_mod/**relm** *[real,private]*
- output_mod/**relna** *[real,private]*
- output_mod/**relz** *[real,private]*
- output_mod/**rmmean** *[real,private]*
- output_mod/**rolmean** *[real,private]*
- output_mod/**timenormlog** *[real,private]*
- output_mod/**timenormrms** *[real,private]*
- output_mod/**timepassedlog** *[real,private]*
- output_mod/**timepassedrms** *[real,private]*

Subroutines and functions

subroutine output_mod/**initialize_output**()

Called from *magic*

Call to *initialize_coeff()*

subroutine output_mod/**finalize_output**()

Called from *magic*

Call to *finalize_coeff()*

subroutine output_mod/**output**(*time*, *tscheme*, *n_time_step*, *l_stop_time*, *l_pot*, *l_log*, *l_graph*, *lrmscalc*, *l_store*, *l_new_rst_file*, *l_spectrum*, *ltocalc*, *ltoframe*, *l_frame*, *n_frame*, *l_cmb*, *n_cmb_sets*, *l_r*, *lorentz_torque_ic*, *lorentz_torque_ma*, *dbdt_cmb_lmloc*, *helasr*, *hel2asr*, *helnaasr*, *helna2asr*, *heleaasr*, *viscasr*, *uhasr*, *duhasr*, *gradsasr*, *fconvasr*, *fkinasr*, *fviscasr*, *fpoynasr*, *fresasr*, *eperpasr*, *eparasr*, *eperpaxiasr*, *eparaxiasr*, *ekinsr*, *ekinlr*, *volsr*)

This subroutine controls most of the output.

Parameters

- **time** *[real,in]* :: Rm (Re) :",RmMean, &
- **tscheme** *[real]*
- **n_time_step** *[integer,in]*
- **l_stop_time** *[logical,in]*
- **l_pot** *[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]*
- **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_cmb_lmloc** (1 - *llmmag* + *ulmmag*) *[complex,in]*
- **helasr** (2,1 - *nrstart* + *nrstop*) *[real,in]*
- **hel2asr** (2,1 - *nrstart* + *nrstop*) *[real,in]*
- **helnaasr** (2,1 - *nrstart* + *nrstop*) *[real,in]*
- **helna2asr** (2,1 - *nrstart* + *nrstop*) *[real,in]*
- **heleaasr** (1 - *nrstart* + *nrstop*) *[real,in]*
- **viscasr** (1 - *nrstart* + *nrstop*) *[real,in]*
- **uhasr** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **duhasr** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **gradsasr** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **fconvasr** (1 - *nrstart* + *nrstop*) *[real,in]*
- **fkinasr** (1 - *nrstart* + *nrstop*) *[real,in]*
- **fviscasr** (1 - *nrstart* + *nrstop*) *[real,in]*
- **fpoynasr** (1 - *nrstartmag* + *nrstopmag*) *[real,in]*
- **fresasr** (1 - *nrstartmag* + *nrstopmag*) *[real,in]*
- **eperpasr** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **eparasr** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **eperpaxiasr** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **eparaxiasr** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **ekinsr** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **ekinlr** (1 - *nrstart* + *nrstop*) *[real,inout]*
- **volsr** (1 - *nrstart* + *nrstop*) *[real,inout]*

Called from *step_time()*

Call to *write_rot()*, *get_e_kin()*, *get_e_mag()*, *get_amplitude()*, *spectrum()*, *spectrum_temp()*, *fields_average()*, *get_power()*, *get_u_square()*, *getdlm()*, *outpar()*, *outperppar()*, *outheat()*, *outhelicity()*, *outphase()*, *outgeos()*, *outto()*, *get_dtblmfinish()*, *zerorms()*, *dtvrms()*, *dtbrms()*, *write_bcmb()*, *write_coeffs()*, *write_pot_mpi()*, *rbrspec()*, *rbpspec()*, *store_mpi()*, *gather_from_lo_to_rank0()*, *gather_all_from_lo_to_rank0()*, *movie_gather_frames_to_rank0()*, *store_movie_frame_ic()*, *logwrite()*, *write_movie_frame()*, *graphout_ic()*, *rint_r()*, *outomega()*

10.17.2 kinetic_energy.f90

Quick access

Variables *e_kin_file*, *n_e_kin_file*, *n_u_square_file*, *u_square_file*

Routines *finalize_kinetic_energy()*, *get_e_kin()*, *get_u_square()*, *initialize_kinetic_energy()*

Needed modules

- *parallel_mod*: This module contains the blocking information
- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *communications* (*get_global_sum()*): This module contains the different MPI communicators used in MagIC.
- *truncation* (*n_r_max()*, *l_max()*): This module defines the grid points and the truncation
- *radial_functions* (*r()*, *or1()*, rscheme_oc(), *or2()*, *r_cmb()*, *r_icb()*, *orho1()*, *orho2()*, *sigma()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *physical_parameters* (*prmag()*, *ek()*, *nvarcond()*): Module containing the physical parameters
- *num_param* (*tscale()*, *escale()*): Module containing numerical and control parameters
- *blocking* (*lo_map()*, *st_map()*, *llm()*, *ulm()*): Module containing blocking information
- *horizontal_data* (*dlh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *logic* (*l_save_out()*, *l_non_rot()*, *l_anel()*): Module containing the logicals that control the run
- *output_data* (*tag()*): This module contains the parameters for output control
- *constants* (*pi()*, *vol_oc()*, *one()*, *two()*, *three()*, *half()*, *four()*, *osq4pi()*): module containing constants and parameters used in the code.
- *integration* (*rint_r()*): Radial integration functions
- *useful* (*cc2real()*): This module contains several useful routines.

Variables

- kinetic_energy/**e_kin_file** *[character,private]*
- kinetic_energy/**e_p_asa** (*) *[real,private/allocatable]*
- kinetic_energy/**e_pa** (*) *[real,private/allocatable]*
- kinetic_energy/**e_t_asa** (*) *[real,private/allocatable]*
- kinetic_energy/**e_ta** (*) *[real,private/allocatable]*
- kinetic_energy/**n_e_kin_file** *[integer,private]*
- kinetic_energy/**n_u_square_file** *[integer,private]*

- kinetic_energy/**u_square_file** *[character,private]*

Subroutines and functions

subroutine kinetic_energy/**initialize_kinetic_energy**()

Called from *magic*

subroutine kinetic_energy/**finalize_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*])

Calculates kinetic energy = 1/2 Integral (v^2 dV). Integration in theta,phi is handled by summation of spherical harmonics Integration in r by using Chebyshev integrals or Simpson rules if FD are used.

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** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]* :: Array containing kinetic field poloidal potential
- **dw** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]* :: Array containing radial derivative of w
- **z** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]* :: Array containing kinetic field toroidal potential
- **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]* :: 1,2,3,4,5
- **ekinr** (*n_r_max*) *[real,out,]* :: Radial profile of kinetic energy

Called from *fields_average()*, *output()*

Call to *cc2real()*, *rint_r()*

subroutine kinetic_energy/**get_u_square**(*time*, *w*, *dw*, *z*, *rolr*)

Calculates square velocity = 1/2 Integral (v^2 dV) Writes the different contributions in u_square.TAG file

Parameters

- **time** *[real,in]* :: 1,2,3, 4,5
- **w** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]* :: Array containing kinetic field poloidal potential
- **dw** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]* :: Array containing radial derivative of w

- **z** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]* :: Array containing kinetic field toroidal potential
- **rolr** (*n_r_max*) *[real,out]* :: local Rossby number

Called from *output()*

Call to *cc2real()*, *rint_r()*

10.17.3 magnetic_energy.f90

Description

This module handles the computation and the writing of the diagnostic files related to magnetic energy: e_mag_oc.TAG, e_mag_ic.TAG, dipole.TAG, eMagR.TAG

Quick access

Variables *bcmb*, *dipole_file*, *e_dipa*, *e_mag_ic_file*, *e_mag_oc_file*, *e_p_asa*, *e_pa*, *e_t_asa*, *e_ta*, *earth_compliance_file*, *lm_max_comp*, *n_compliance_file*, *n_dipole_file*, *n_e_mag_ic_file*, *n_e_mag_oc_file*, *n_phi_max_comp*, *n_theta_max_comp*, *plm_comp*

Routines *finalize_magnetic_energy()*, *get_br_skew()*, *get_e_mag()*, *initialize_magnetic_energy()*

Needed modules

- *parallel_mod*: This module contains the blocking information
- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *truncation* (*n_r_maxmag()*, *n_r_ic_maxmag()*, *n_r_max()*, *n_r_ic_max()*, *lm_max()*, *minc()*): This module defines the grid points and the truncation
- *radial_data* (*n_r_cmb()*): This module defines the MPI decomposition in the radial direction.
- *radial_functions* (*r_icb()*, *r_cmb()*, *r_ic()*, *dr_fac_ic()*, *chebt_ic()*, *sigma()*, *orho1()*, *r()*, *or2()*, `rscheme_oc()`): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *physical_parameters* (*lffac()*, *kbotb()*, *ktopb()*): Module containing the physical parameters
- *num_param* (*escale()*, *tscale()*): Module containing numerical and control parameters
- *blocking* (*st_map()*, *lo_map()*, *llmmag()*, *ulmmag()*): Module containing blocking information
- *horizontal_data* (*dlh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *logic* (*l_cond_ic()*, *l_mag()*, *l_mag_lf()*, *l_save_out()*, *l_earth_likeness()*, *l_full_sphere()*): Module containing the logicals that control the run
- *movie_data* (*moviedipcolat()*, *moviediplon()*, *moviedipstrength()*, *moviedipstrengthgeo()*)
- *output_data* (*tag()*, *l_max_comp()*, *l_geo()*): This module contains the parameters for output control

- *constants* (*pi()*, *zero()*, *one()*, *two()*, *half()*, *four()*, *osq4pi()*): module containing constants and parameters used in the code.
- *special* (*n_imp()*, *rrmp()*): This module contains all variables for the case of an imposed IC dipole, an imposed external magnetic field and a special boundary forcing to excite inertial modes
- *integration* (*rint_r()*, *rintic()*): Radial integration functions
- *useful* (*cc2real()*, *cc22real()*): This module contains several useful routines.
- *plms_theta* (*plm_theta()*)
- *communications* (*gather_from_lo_to_rank0()*, *reduce_radial()*, *reduce_scalar()*, *send_lm_pair_to_master()*): This module contains the different MPI communicators used in MagIC.

Variables

- magnetic_energy/**bcmb** (*) *[complex,private/allocatable]*
- magnetic_energy/**dipole_file** *[character,private]*
- magnetic_energy/**e_dipa** (*) *[real,private/allocatable]*
  Time-averaged dipole (l=1) energy
- magnetic_energy/**e_mag_ic_file** *[character,private]*
- magnetic_energy/**e_mag_oc_file** *[character,private]*
- magnetic_energy/**e_p_asa** (*) *[real,private/allocatable]*
  Time-averaged axisymmetric poloidal energy
- magnetic_energy/**e_pa** (*) *[real,private/allocatable]*
  Time-averaged poloidal energy
- magnetic_energy/**e_t_asa** (*) *[real,private/allocatable]*
  Time-averaged axisymmetric toroidal energy
- magnetic_energy/**e_ta** (*) *[real,private/allocatable]*
  Time-averaged toroidal energy
- magnetic_energy/**earth_compliance_file** *[character,private]*
- magnetic_energy/**lm_max_comp** *[integer,private]*
- magnetic_energy/**n_compliance_file** *[integer,private]*
- magnetic_energy/**n_dipole_file** *[integer,private]*
- magnetic_energy/**n_e_mag_ic_file** *[integer,private]*
- magnetic_energy/**n_e_mag_oc_file** *[integer,private]*
- magnetic_energy/**n_phi_max_comp** *[integer,private]*
- magnetic_energy/**n_theta_max_comp** *[integer,private]*
- magnetic_energy/**plm_comp** (*,*) *[real,private/allocatable]*

Subroutines and functions

subroutine magnetic_energy/**initialize_magnetic_energy**()
Open diagnostic files and allocate memory

Called from *magic*

Call to *plm_theta()*

subroutine magnetic_energy/**finalize_magnetic_energy**()
Close file and deallocates global arrays

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 Chebyshev integrals or Simpson rule depending whether FD or Cheb is used.

Parameters

- **time** *[real,in]* :: 1
- **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** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]* :: Array containing magnetic field poloidal potential
- **db** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]* :: Array containing radial derivative of b
- **aj** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]* :: Array containing magnetic field toroidal potential
- **b_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,in]* :: Array containing IC magnetic field poloidal potential
- **db_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,in]* :: Array containing radial derivative of IC b
- **aj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,in]* :: Array containing IC magnetic field toroidal potential
- **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]* :: 4,5
- **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]* :: 8,9
- **e_cmb** *[real,out]* :: 17
- **dip** *[real,out]* :: 4
- **dipcmb** *[real,out]* :: 6
- **elsanel** *[real,out]* :: Radially averaged Elsasser number

Called from *fields_average()*, *output()*

Call to *cc2real()*, *reduce_scalar()*, *gather_from_lo_to_rank0()*, *rint_r()*, *cc22real()*, *rintic()*, *get_br_skew()*

subroutine magnetic_energy/**get_br_skew**(*bcmb*, *br_skew*)

This subroutine calculates the skewness of the radial magnetic field at the outer boundary. bskew := <B^4> / <B^2>^2 where <..> is average over the surface

Parameters

- **bcmb** (*lm_max*) *[complex,in]*
- **br_skew** *[real,out]*

Called from *get_e_mag()*

10.17.4 getDlm.f90

Description

This module is used to calculate the lengthscales

Quick access

Routines *getdlm()*

Needed modules

- *parallel_mod*: This module contains the blocking information
- *precision_mod*: This module controls the precision used in MagIC
- *communications* (*reduce_radial()*): This module contains the different MPI communicators used in MagIC.
- *truncation* (*minc()*, *m_max()*, *l_max()*, *n_r_max()*): This module defines the grid points and the truncation
- *radial_functions* (*or2()*, *r()*, rscheme_oc(), *orho1()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *num_param* (*escale()*): Module containing numerical and control parameters
- *blocking* (*lo_map()*, *st_map()*, *llm()*, *ulm()*): Module containing blocking information

- *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.
- *useful* (*abortrun()*): This module contains several useful routines.
- *integration* (*rint_r()*): Radial integration functions

Variables

Subroutines and functions

subroutine getdlm_mod/**getdlm**(*w*, *dw*, *z*, *dl*, *dlr*, *dm*, *dlc*, *dlpolpeak*, *dlrc*, *dlpolpeakr*, *switch_bn*)

This routine is used to compute integral lengthscale using spectra

Parameters

- **w** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **dw** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **z** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **dl** *[real,out]*
- **dlr** (*n_r_max*) *[real,out]*
- **dm** *[real,out]*
- **dlc** *[real,out]*
- **dlpolpeak** *[real,out]*
- **dlrc** (*n_r_max*) *[real,out]*
- **dlpolpeakr** (*n_r_max*) *[real,out]*
- **switch_bn** *[character,in]*

Called from *output()*

Call to *cc2real()*, *abortrun()*, *rint_r()*

10.17.5 outMisc.f90

Description

This module contains several subroutines that can compute and store various informations: helicity, heat transfer.

Quick access

Variables *heat_file*, *helicity_file*, *n_heat_file*, *n_helicity_file*, *n_phase_file*, *n_rmelt_file*, *phase_file*, *phimeanr*, *pmeanr*, *rhomeanr*, *rmelt_file*, *smeanr*, *tmeanr*, *tphi*, *tphiold*, *ximeanr*

Routines *finalize_outmisc_mod()*, *initialize_outmisc_mod()*, *outheat()*, *outhelicity()*, *outphase()*

Needed modules

- *parallel_mod*: This module contains the blocking information
- *precision_mod*: This module controls the precision used in MagIC
- *communications* (*gather_from_rloc()*): This module contains the different MPI communicators used in MagIC.
- *truncation* (*l_max()*, *n_r_max()*, *lm_max()*, *nlat_padded()*, *n_theta_max()*): This module defines the grid points and the truncation
- *radial_data* (*n_r_icb()*, *n_r_cmb()*, *nrstart()*, *nrstop()*): This module defines the MPI decomposition in the radial direction.
- *radial_functions* (*r_icb()*, rscheme_oc(), *kappa()*, *r_cmb()*, *temp0()*, *r()*, *rho0()*, *dltemp0()*, *dlalpha0()*, *beta()*, *orho1()*, *alpha0()*, *otemp1()*, *ogrun()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *physical_parameters* (*vischeatfac()*, *thexpnb()*, *opr()*, *stef()*): Module containing the physical parameters
- *num_param* (*lscale()*, *escale()*): Module containing numerical and control parameters
- *blocking* (*llm()*, *ulm()*, *lo_map()*): Module containing blocking information
- *mean_sd* (mean_sd_type()): This module contains a small type that simply handles two arrays (mean and SD) This type is used for time-averaged outputs (and their standard deviations).
- *horizontal_data* (*gauss()*, *theta_ord()*, *n_theta_cal2ord()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *logic* (*l_save_out()*, *l_anelastic_liquid()*, *l_heat()*, *l_hel()*, *l_temperature_diff()*, *l_chemical_conv()*, *l_phase_field()*): Module containing the logicals that control the run
- *output_data* (*tag()*): This module contains the parameters for output control
- *constants* (*pi()*, *vol_oc()*, *osq4pi()*, *sq4pi()*, *one()*, *two()*, *four()*, *half()*, *zero()*): module containing constants and parameters used in the code.
- *start_fields* (*topcond()*, *botcond()*, *deltacond()*, *topxicond()*, *botxicond()*, *deltaxicond()*): This module is used to set-up the initial starting fields. They can be obtained by reading a starting checkpoint file or by setting some starting conditions.
- *useful* (*cc2real()*, *round_off()*): This module contains several useful routines.
- *integration* (*rint_r()*): Radial integration functions
- *sht* (*axi_to_spat()*)

Variables

- outmisc_mod/**heat_file** *[character,private]*
- outmisc_mod/**helicity_file** *[character,private]*
- outmisc_mod/**n_calls** *[integer,private]*
- outmisc_mod/**n_heat_file** *[integer,private]*
- outmisc_mod/**n_helicity_file** *[integer,private]*
- outmisc_mod/**n_phase_file** *[integer,private]*
- outmisc_mod/**n_rmelt_file** *[integer,private]*
- outmisc_mod/**phase_file** *[character,private]*
- outmisc_mod/**phimeanr** *[mean_sd_type,private]*
- outmisc_mod/**pmeanr** *[mean_sd_type,private]*
- outmisc_mod/**rhomeanr** *[mean_sd_type,private]*
- outmisc_mod/**rmelt_file** *[character,private]*
- outmisc_mod/**smeanr** *[mean_sd_type,private]*
- outmisc_mod/**tmeanr** *[mean_sd_type,private]*
- outmisc_mod/**tphi** *[real,private]*
- outmisc_mod/**tphiold** *[real,private]*
- outmisc_mod/**ximeanr** *[mean_sd_type,private]*

Subroutines and functions

subroutine outmisc_mod/**initialize_outmisc_mod**()

This subroutine handles the opening the output diagnostic files that have to do with heat transfer or helicity.

Called from *magic*

subroutine outmisc_mod/**finalize_outmisc_mod**()

This subroutine handles the closing of the time series of heat.TAG, hel.TAG and phase.TAG

Called from *magic*

subroutine outmisc_mod/**outhelicity**(*timescaled*, *helasr*, *hel2asr*, *helnaasr*, *helna2asr*, *heleaasr*)

This subroutine is used to store informations about kinetic helicity

Parameters

- **timescaled** *[real,in]*
- **helasr** (2,1 - *nrstart* + *nrstop*) *[real,in]*
- **hel2asr** (2,1 - *nrstart* + *nrstop*) *[real,in]*
- **helnaasr** (2,1 - *nrstart* + *nrstop*) *[real,in]*
- **helna2asr** (2,1 - *nrstart* + *nrstop*) *[real,in]*
- **heleaasr** (1 - *nrstart* + *nrstop*) *[real,in]*

Called from *output()*

Call to *gather_from_rloc()*, *rint_r()*

subroutine outmisc_mod/**outheat**(*time*, *timepassed*, *timenorm*, *l_stop_time*, *s*, *ds*, *p*, *dp*, *xi*, *dxi*)

This subroutine is used to store informations about heat transfer (i.e. Nusselt number, temperature, entropy, ...)

Parameters

- **time** *[real,in]*
- **timepassed** *[real,in]*
- **timenorm** *[real,in]*
- **l_stop_time** *[logical,in]*
- **s** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **ds** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **p** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **dp** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **xi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **dxi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*

Called from *output()*

Call to *rint_r()*, *round_off()*

subroutine outmisc_mod/**outphase**(*time*, *timepassed*, *timenorm*, *l_stop_time*, *nlogs*, *s*, *ds*, *phi*, *ekinsr*, *ekinlr*, *volsr*)

This subroutine handles the writing of time series related with phase field: phase.TAG

Parameters

- **time** *[real,in]* :: Time
- **timepassed** *[real,in]* :: Time passed since last call
- **timenorm** *[real,in]*
- **l_stop_time** *[logical,in]* :: Last iteration
- **nlogs** *[integer,in]* :: Number of log outputs
- **s** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]* :: Entropy/Temperature
- **ds** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]* :: Radial der. of Entropy/Temperature
- **phi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]* :: Phase field
- **ekinsr** (1 - *nrstart* + *nrstop*) *[real,in]* :: Kinetic energy in solidus
- **ekinlr** (1 - *nrstart* + *nrstop*) *[real,in]* :: Kinetic energy in liquidus
- **volsr** (1 - *nrstart* + *nrstop*) *[real,in]* :: Volume of the solid phase

Called from *output()*

Call to *gather_from_rloc()*, *axi_to_spat()*, *rint_r()*, *round_off()*

10.17.6 outRot.f90

Description

This module handles the writing of several diagnostic files related to the rotation: angular momentum (AM.TAG), drift (drift.TAG), inner core and mantle rotations.

Quick access

Variables *angular_file*, *driftbd_file*, *driftbq_file*, *driftvd_file*, *driftvq_file*, *inerp_file*, *inert_file*, *n_angular_file*, *n_driftbd_file*, *n_driftbq_file*, *n_driftvd_file*, *n_driftvq_file*, *n_inerp_file*, *n_inert_file*, *n_rot_file*, *n_sric_file*, *n_srma_file*, *rot_file*, *sric_file*, *srma_file*

Routines *finalize_outrot()*, *get_angular_moment()*, *get_angular_moment_rloc()*, *get_lorentz_torque()*, *get_viscous_torque()*, *initialize_outrot()*, *write_rot()*

Needed modules

- *parallel_mod*: This module contains the blocking information
- *precision_mod*: This module controls the precision used in MagIC
- *communications* (*allgather_from_rloc()*, *send_lm_pair_to_master()*): This module contains the different MPI communicators used in MagIC.
- *truncation* (*n_r_max()*, *n_r_maxmag()*, *minc()*, *n_phi_max()*, *n_theta_max()*): This module defines the grid points and the truncation
- *radial_data* (*n_r_cmb()*, *n_r_icb()*, *nrstart()*, *nrstop()*): This module defines the MPI decomposition in the radial direction.
- *radial_functions* (*r_icb()*, *r_cmb()*, *r()*, rscheme_oc(), *beta()*, *visc()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *physical_parameters* (*kbotv()*, *ktopv()*, *lffac()*): Module containing the physical parameters
- *num_param* (*lscale()*, *tscale()*, *vscale()*): Module containing numerical and control parameters
- *blocking* (*lo_map()*, *lm_balance()*, *llm()*, *ulm()*, *llmmag()*, *ulmmag()*): Module containing blocking information
- *logic* (*l_am()*, *l_save_out()*, *l_iner()*, *l_sric()*, *l_rot_ic()*, *l_srma()*, *l_rot_ma()*, *l_mag_lf()*, *l_mag()*, *l_drift()*, *l_finite_diff()*, *l_full_sphere()*): Module containing the logicals that control the run
- *output_data* (*tag()*): This module contains the parameters for output control
- *constants* (*c_moi_oc()*, *c_moi_ma()*, *c_moi_ic()*, *pi()*, *y11_norm()*, *y10_norm()*, *zero()*, *two()*, *third()*, *four()*, *half()*): module containing constants and parameters used in the code.
- *integration* (*rint_r()*): Radial integration functions
- *horizontal_data* (*costheta()*, *gauss()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *special* (*bic()*, *lgrenoble()*): This module contains all variables for the case of an imposed IC dipole, an imposed external magnetic field and a special boundary forcing to excite inertial modes
- *useful* (*abortrun()*): This module contains several useful routines.

Variables

- outrot/**angular_file** *[character,private]*
- outrot/**driftbd_file** *[character,private]*
- outrot/**driftbq_file** *[character,private]*
- outrot/**driftvd_file** *[character,private]*
- outrot/**driftvq_file** *[character,private]*
- outrot/**inerp_file** *[character,private]*
- outrot/**inert_file** *[character,private]*
- outrot/**n_angular_file** *[integer,private]*
- outrot/**n_driftbd_file** *[integer,private]*
- outrot/**n_driftbq_file** *[integer,private]*
- outrot/**n_driftvd_file** *[integer,private]*
- outrot/**n_driftvq_file** *[integer,private]*
- outrot/**n_inerp_file** *[integer,private]*
- outrot/**n_inert_file** *[integer,private]*
- outrot/**n_rot_file** *[integer,private]*
- outrot/**n_sric_file** *[integer,private]*
- outrot/**n_srma_file** *[integer,private]*
- outrot/**rot_file** *[character,private]*
- outrot/**sric_file** *[character,private]*
- outrot/**srma_file** *[character,private]*

Subroutines and functions

subroutine outrot/**initialize_outrot**()

Called from *magic*

subroutine outrot/**finalize_outrot**()

Called from *magic*

subroutine outrot/**write_rot**(*time*, *dt*, *ekinic*, *ekinma*, *w*, *z*, *dz*, *b*, *omega_ic*, *omega_ma*, *lorentz_torque_ic*, *lorentz_torque_ma*)

– Input of variables:

Parameters

- **time** *[real,in]*
- **dt** *[real,in]*
- **ekinic** *[real,out]*
- **ekinma** *[real,out]*

- **w** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **z** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **dz** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **b** (1 - *llmmag* + *ulmmag*,*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 *get_viscous_torque()*, *get_angular_moment()*

subroutine outrot/**get_viscous_torque**(*viscous_torque*, *z10*, *dz10*, *r*, *dlrho*, *nu*)

Purpose of this subroutine is to calculate the viscous torque on mantle or inner core respectively.

$$\Gamma_\nu = 4\sqrt{\pi/3}\nu r \left[\frac{\partial z_{10}}{\partial r} - (\frac{2}{r} + \beta) z_{10}\right]$$

Parameters

- **viscous_torque** *[real,out]*
- **z10** *[real,in]*
- **dz10** *[real,in]* :: z10 coefficient and its radial deriv.
- **r** *[real,in]* :: radius (ICB or CMB)
- **dlrho** *[real,in]* :: dln(rho)/dr
- **nu** *[real,in]* :: viscosity

Called from *write_rot()*, *get_power()*

subroutine outrot/**get_lorentz_torque**(*lorentz_torque*, *br*, *bp*, *nr*)

Purpose of this subroutine is to calculate the Lorentz torque on mantle or inner core respectively.

Note: `lorentz_torque` must be set to zero before loop over theta blocks is started.

Warning: subroutine returns `-lorentz_torque` if used at CMB to calculate torque on mantle because if the inward surface normal vector.

The Prandtl number is always the Prandtl number of the outer core. This comes in via scaling of the magnetic field. Theta alternates between northern and southern hemisphere in `br` and `bp` but not in gauss. This has to be cared for, and we use: `gauss(latitude)=gauss(-latitude)` here.

Parameters

- **lorentz_torque** *[real,inout]* :: Lorentz torque
- **br** (,) *[real,in]* :: array containing $r^2 B_r$

- **bp** (,) *[real,in]* :: array containing $r \sin\theta B_\phi$
- **nr** *[integer,in]* :: radial level

Called from *radialloop()*

subroutine outrot/**get_angular_moment**(*z10*, *z11*, *omega_ic*, *omega_ma*, *angular_moment_oc*, *angular_moment_ic*, *angular_moment_ma*)

Calculates angular momentum of outer core, inner core and mantle. For outer core we need `z(l=1|m=0,1|r)`, for inner core and mantle the respective rotation rates are needed.

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 *write_rot()*, *get_tor_rhs_imp()*

Call to *rint_r()*

subroutine outrot/**get_angular_moment_rloc**(*z10*, *z11*, *omega_ic*, *omega_ma*, *angular_moment_oc*, *angular_moment_ic*, *angular_moment_ma*)

Calculates angular momentum of outer core, inner core and mantle. For outer core we need `z(l=1|m=0,1|r)`, for inner core and mantle the respective rotation rates are needed. This is the version that takes r-distributed arrays as input arrays.

Parameters

- **z10** (1 - *nrstart* + *nrstop*) *[complex,in]*
- **z11** (1 - *nrstart* + *nrstop*) *[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 *get_tor_rhs_imp_ghost()*

Call to *allgather_from_rloc()*, *rint_r()*

10.17.7 outPar.f90

Description

This module is used to compute several time-averaged radial profiles: fluxes, boundary layers, etc.

Quick access

Variables *dlpolpeak*, *dlv*, *dlvc*, *duh*, *entropy*, *epar*, *eparaxi*, *eperp*, *eperpaxi*, *fcond*, *fconv*, *fkin*, *fpoyn*, *fres*, *fvisc*, *gradt2*, *n_perppar_file*, *perppar_file*, *rm*, *rol*, *uh*, *urol*

Routines *finalize_outpar_mod()*, *initialize_outpar_mod()*, *outpar()*, *outperppar()*

Needed modules

- *parallel_mod*: This module contains the blocking information
- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *communications* (*gather_from_rloc()*): This module contains the different MPI communicators used in MagIC.
- *truncation* (*n_r_max()*, *n_r_maxmag()*, *l_max()*, *lm_max()*, *l_maxmag()*): This module defines the grid points and the truncation
- *logic* (*l_viscbccalc()*, *l_anel()*, *l_fluxprofs()*, *l_mag_nl()*, *l_perppar()*, *l_save_out()*, *l_temperature_diff()*, *l_anelastic_liquid()*): Module containing the logicals that control the run
- *horizontal_data* (*gauss()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *fields* (*s_rloc()*, *ds_rloc()*, *p_rloc()*, *dp_rloc()*): This module contains all the fields used in MagIC in the hybrid (LM,r) space as well as their radial derivatives. It defines both the LM-distributed arrays and the R-distributed arrays....
- *physical_parameters* (*ek()*, *prmag()*, *ohmlossfac()*, *vischeatfac()*, *opr()*, *kbots()*, *ktops()*, *thexpnb()*, *ekscaled()*): Module containing the physical parameters
- *num_param* (*tscale()*): Module containing numerical and control parameters
- *constants* (*pi()*, *mass()*, *osq4pi()*, *sq4pi()*, *half()*, *two()*, *four()*): module containing constants and parameters used in the code.
- *radial_functions* (*r()*, *or2()*, *sigma()*, *rho0()*, *kappa()*, *temp0()*, rscheme_oc(), *orho1()*, *dlalpha0()*, *dltemp0()*, *beta()*, *alpha0()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *radial_data* (*n_r_icb()*, *nrstart()*, *nrstop()*, *nrstartmag()*, *nrstopmag()*): This module defines the MPI decomposition in the radial direction.
- *output_data* (*tag()*): This module contains the parameters for output control
- *useful* (*cc2real()*, *round_off()*): This module contains several useful routines.
- *mean_sd* (mean_sd_type()): This module contains a small type that simply handles two arrays (mean and SD) This type is used for time-averaged outputs (and their standard deviations).
- *integration* (*rint_r()*): Radial integration functions

Variables

- outpar_mod/**dlpolpeak** *[mean_sd_type,private]*
- outpar_mod/**dlv** *[mean_sd_type,private]*
- outpar_mod/**dlvc** *[mean_sd_type,private]*
- outpar_mod/**duh** *[mean_sd_type,private]*
- outpar_mod/**entropy** *[mean_sd_type,private]*
- outpar_mod/**epar** *[mean_sd_type,private]*
- outpar_mod/**eparaxi** *[mean_sd_type,private]*
- outpar_mod/**eperp** *[mean_sd_type,private]*
- outpar_mod/**eperpaxi** *[mean_sd_type,private]*
- outpar_mod/**fcond** *[mean_sd_type,private]*
- outpar_mod/**fconv** *[mean_sd_type,private]*
- outpar_mod/**fkin** *[mean_sd_type,private]*
- outpar_mod/**fpoyn** *[mean_sd_type,private]*
- outpar_mod/**fres** *[mean_sd_type,private]*
- outpar_mod/**fvisc** *[mean_sd_type,private]*
- outpar_mod/**gradt2** *[mean_sd_type,private]*
- outpar_mod/**n_calls** *[integer,private]*
- outpar_mod/**n_perppar_file** *[integer,private]*
- outpar_mod/**perppar_file** *[character,private]*
- outpar_mod/**rm** *[mean_sd_type,private]*
- outpar_mod/**rol** *[mean_sd_type,private]*
- outpar_mod/**uh** *[mean_sd_type,private]*
- outpar_mod/**urol** *[mean_sd_type,private]*

Subroutines and functions

subroutine outpar_mod/**initialize_outpar_mod**()
Memory allocation and file openings

Called from *magic*

subroutine outpar_mod/**finalize_outpar_mod**()

Called from *magic*

subroutine outpar_mod/**outpar**(*timepassed*, *timenorm*, *l_stop_time*, *ekinr*, *rolru2*, *dlvr*, *dlvrc*, *dlpolpeakr*, *uhasr*, *duhasr*, *gradt2asr*, *fconvasr*, *fkinasr*, *fviscasr*, *fpoynasr*, *fresasr*, *rmr*)
— Input of variables

Parameters

- **timepassed** *[real,in]*

- **timenorm** *[real,in]*
- **l_stop_time** *[logical,in]*
- **ekinr** (*n_r_max*) *[real,in]* :: kinetic energy w radius
- **rolru2** (*n_r_max*) *[real,in]*
- **dlvr** (*n_r_max*) *[real,in]*
- **dlvrc** (*n_r_max*) *[real,in]*
- **dlpolpeakr** (*n_r_max*) *[real,in]*
- **uhasr** (1 - *nrstart* + *nrstop*) *[real,inout]* :: Normalisation for the theta integration
- **duhasr** (1 - *nrstart* + *nrstop*) *[real,inout]* :: Normalisation for the theta integration
- **gradt2asr** (1 - *nrstart* + *nrstop*) *[real,inout]* :: Normalisation for the theta integration
- **fconvasr** (1 - *nrstart* + *nrstop*) *[real,in]*
- **fkinasr** (1 - *nrstart* + *nrstop*) *[real,in]*
- **fviscasr** (1 - *nrstart* + *nrstop*) *[real,in]*
- **fpoynasr** (1 - *nrstartmag* + *nrstopmag*) *[real,in]*
- **fresasr** (1 - *nrstartmag* + *nrstopmag*) *[real,in]*
- **rmr** (*n_r_max*) *[real,out]*

Called from *output()*

Call to *gather_from_rloc()*, *round_off()*

subroutine outpar_mod/**outerppar**(*time*, *timepassed*, *timenorm*, *l_stop_time*, *eperpasr*, *eparasr*, *eperpaxiasr*, *eparaxiasr*)

Parameters

- **time** *[real,in]*
- **timepassed** *[real,in]*
- **timenorm** *[real,in]*
- **l_stop_time** *[logical,in]*
- **eperpasr** (1 - *nrstart* + *nrstop*) *[real,inout]* :: Normalisation for the theta integration
- **eparasr** (1 - *nrstart* + *nrstop*) *[real,inout]* :: Normalisation for the theta integration
- **eperpaxiasr** (1 - *nrstart* + *nrstop*) *[real,inout]* :: Normalisation for the theta integration
- **eparaxiasr** (1 - *nrstart* + *nrstop*) *[real,inout]* :: Normalisation for the theta integration

Called from *output()*

Call to *gather_from_rloc()*, *rint_r()*, *round_off()*

10.17.8 power.f90

Description

This module handles the writing of the power budget

Quick access

Variables *buo_ave*, *buo_chem_ave*, *ediffint*, *n_calls*, *n_power_file*, *ohm_ave*, *power_file*, *powerdiff*, *visc_ave*

Routines *finalize_output_power()*, *get_power()*, *initialize_output_power()*

Needed modules

- *parallel_mod*: This module contains the blocking information
- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *communications* (*gather_from_rloc()*, *reduce_radial()*, *send_lm_pair_to_master()*): This module contains the different MPI communicators used in MagIC.
- *truncation* (*n_r_ic_maxmag()*, *n_r_max()*, *n_r_ic_max()*, *l_max()*, *n_r_maxmag()*): This module defines the grid points and the truncation
- *radial_data* (*n_r_icb()*, *n_r_cmb()*, *nrstart()*, *nrstop()*): This module defines the MPI decomposition in the radial direction.
- *radial_functions* (*r_cmb()*, *r_icb()*, *r()*, rscheme_oc(), *chebt_ic()*, *or2()*, *o_r_ic2()*, *lambda()*, *temp0()*, *o_r_ic()*, *rgrav()*, *r_ic()*, *dr_fac_ic()*, *alpha0()*, *orho1()*, *otemp1()*, *beta()*, *visc()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *physical_parameters* (*kbotv()*, *ktopv()*, *opm()*, *lffac()*, *buofac()*, *chemfac()*, *thexpnb()*, *vischeatfac()*): Module containing the physical parameters
- *num_param* (*tscale()*, *escale()*): Module containing numerical and control parameters
- *blocking* (*lo_map()*, *st_map()*, *llm()*, *ulm()*, *llmmag()*, *ulmmag()*): Module containing blocking information
- *horizontal_data* (*dlh()*, *gauss()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *logic* (*l_rot_ic()*, *l_sric()*, *l_rot_ma()*, *l_srma()*, *l_save_out()*, *l_conv()*, *l_cond_ic()*, *l_heat()*, *l_mag()*, *l_chemical_conv()*, *l_anelastic_liquid()*): Module containing the logicals that control the run
- *output_data* (*tag()*): This module contains the parameters for output control
- *mean_sd* (mean_sd_type()): This module contains a small type that simply handles two arrays (mean and SD) This type is used for time-averaged outputs (and their standard deviations).
- *useful* (*cc2real()*, *cc22real()*, *round_off()*): This module contains several useful routines.
- *integration* (*rint_r()*, *rintic()*): Radial integration functions
- *outrot* (*get_viscous_torque()*): This module handles the writing of several diagnostic files related to the rotation: angular momentum (AM.TAG), drift (drift.TAG), inner core and mantle rotations....

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

Variables

- power/**buo_ave** *[mean_sd_type,private]*
- power/**buo_chem_ave** *[mean_sd_type,private]*
- power/**ediffint** *[real,private]*
- power/**n_calls** *[integer,private]*
- power/**n_power_file** *[integer,private]*
- power/**ohm_ave** *[mean_sd_type,private]*
- power/**power_file** *[character,private]*
- power/**powerdiff** *[real,private]*
- power/**visc_ave** *[mean_sd_type,private]*

Subroutines and functions

subroutine power/**initialize_output_power**()
Memory allocation

Called from *magic*

subroutine power/**finalize_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*, *z*, *dz*, *s*, *xi*, *b*, *ddb*, *aj*, *dj*, *db_ic*, *ddb_ic*, *aj_ic*, *dj_ic*, *viscasr*, *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** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **z** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **dz** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **s** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **xi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **b** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]*
- **ddb** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]*
- **aj** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]*
- **dj** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]*
- **db_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,in]*
- **ddb_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,in]*
- **aj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,in]*
- **dj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,in]*
- **viscasr** (1 - *nrstart* + *nrstop*) *[real,in]*
- **viscdiss** *[real,out]*
- **ohmdiss** *[real,out]*

Called from *output()*

Call to *cc2real()*, *gather_from_rloc()*, *rint_r()*, *rintic()*, *get_viscous_torque()*, *round_off()*

10.17.9 spectra.f90

Quick access

Variables *am_kpol_file*, *am_ktor_file*, *am_mpol_file*, *am_mtor_file*, *dt_icb_l_ave*, *dt_icb_m_ave*, *e_kin_p_l_ave*, *e_kin_p_m_ave*, *e_kin_p_r_l_ave*, *e_kin_p_r_m_ave*, *e_kin_t_l_ave*, *e_kin_t_m_ave*, *e_kin_t_r_l_ave*, *e_kin_t_r_m_ave*, *e_mag_cmb_l_ave*, *e_mag_cmb_m_ave*, *e_mag_p_l_ave*, *e_mag_p_m_ave*, *e_mag_p_r_l_ave*, *e_mag_p_r_m_ave*, *e_mag_t_l_ave*, *e_mag_t_m_ave*, *e_mag_t_r_l_ave*, *e_mag_t_r_m_ave*, *n_am_kpol_file*, *n_am_ktor_file*, *n_am_mpol_file*, *n_am_mtor_file*, *t_icb_l_ave*, *t_icb_m_ave*, *t_l_ave*, *t_m_ave*, *u2_p_l_ave*, *u2_p_m_ave*, *u2_t_l_ave*, *u2_t_m_ave*

Routines *finalize_spectra()*, *get_amplitude()*, *initialize_spectra()*, *spectrum()*, *spectrum_temp()*

Needed modules

- *parallel_mod*: This module contains the blocking information
- *precision_mod*: This module controls the precision used in MagIC
- *communications* (*reduce_radial()*): This module contains the different MPI communicators used in MagIC.
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *truncation* (*n_r_max()*, *n_r_ic_maxmag()*, *n_r_maxmag()*, *n_r_ic_max()*, *l_max()*, *minc()*): This module defines the grid points and the truncation
- *radial_data* (*n_r_cmb()*, *n_r_icb()*): This module defines the MPI decomposition in the radial direction.
- *radial_functions* (*orho1()*, *orho2()*, *r_ic()*, *chebt_ic()*, *r()*, *r_cmb()*, `rscheme_oc()`, *or2()*, *r_icb()*, *dr_fac_ic()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *physical_parameters* (*lffac()*): Module containing the physical parameters
- *num_param* (*escale()*, *tscale()*): Module containing numerical and control parameters
- *blocking* (*lo_map()*, *st_map()*, *llm()*, *ulm()*, *llmmag()*, *ulmmag()*): Module containing blocking information
- *horizontal_data* (*dlh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *logic* (*l_mag()*, *l_anel()*, *l_cond_ic()*, *l_heat()*, *l_save_out()*, *l_energy_modes()*, *l_2d_spectra()*): Module containing the logicals that control the run
- *output_data* (*tag()*, *log_file()*, *n_log_file()*, *m_max_modes()*): This module contains the parameters for output control
- *useful* (*cc2real()*, *cc22real()*, *abortrun()*): This module contains several useful routines.
- *integration* (*rint_r()*, *rintic()*): Radial integration functions
- *constants* (*pi()*, *vol_oc()*, *half()*, *one()*, *four()*): module containing constants and parameters used in the code.
- *mean_sd* (`mean_sd_type()`, `mean_sd_2d_type()`): This module contains a small type that simply handles two arrays (mean and SD) This type is used for time-averaged outputs (and their standard deviations).

Variables

- spectra/**am_kpol_file** *[character,private]*
- spectra/**am_ktor_file** *[character,private]*
- spectra/**am_mpol_file** *[character,private]*
- spectra/**am_mtor_file** *[character,private]*
- spectra/**dt_icb_l_ave** *[mean_sd_type,private]*
- spectra/**dt_icb_m_ave** *[mean_sd_type,private]*
- spectra/**e_kin_p_l_ave** *[mean_sd_type,private]*
- spectra/**e_kin_p_m_ave** *[mean_sd_type,private]*

- spectra/**e_kin_p_r_l_ave** *[mean_sd_2d_type,private]*
- spectra/**e_kin_p_r_m_ave** *[mean_sd_2d_type,private]*
- spectra/**e_kin_t_l_ave** *[mean_sd_type,private]*
- spectra/**e_kin_t_m_ave** *[mean_sd_type,private]*
- spectra/**e_kin_t_r_l_ave** *[mean_sd_2d_type,private]*
- spectra/**e_kin_t_r_m_ave** *[mean_sd_2d_type,private]*
- spectra/**e_mag_cmb_l_ave** *[mean_sd_type,private]*
- spectra/**e_mag_cmb_m_ave** *[mean_sd_type,private]*
- spectra/**e_mag_p_l_ave** *[mean_sd_type,private]*
- spectra/**e_mag_p_m_ave** *[mean_sd_type,private]*
- spectra/**e_mag_p_r_l_ave** *[mean_sd_2d_type,private]*
- spectra/**e_mag_p_r_m_ave** *[mean_sd_2d_type,private]*
- spectra/**e_mag_t_l_ave** *[mean_sd_type,private]*
- spectra/**e_mag_t_m_ave** *[mean_sd_type,private]*
- spectra/**e_mag_t_r_l_ave** *[mean_sd_2d_type,private]*
- spectra/**e_mag_t_r_m_ave** *[mean_sd_2d_type,private]*
- spectra/**n_am_kpol_file** *[integer,private]*
- spectra/**n_am_ktor_file** *[integer,private]*
- spectra/**n_am_mpol_file** *[integer,private]*
- spectra/**n_am_mtor_file** *[integer,private]*
- spectra/**t_icb_l_ave** *[mean_sd_type,private]*
- spectra/**t_icb_m_ave** *[mean_sd_type,private]*
- spectra/**t_l_ave** *[mean_sd_type,private]*
- spectra/**t_m_ave** *[mean_sd_type,private]*
- spectra/**u2_p_l_ave** *[mean_sd_type,private]*
- spectra/**u2_p_m_ave** *[mean_sd_type,private]*
- spectra/**u2_t_l_ave** *[mean_sd_type,private]*
- spectra/**u2_t_m_ave** *[mean_sd_type,private]*

Subroutines and functions

subroutine spectra/**initialize_spectra**()

Called from *magic*

subroutine spectra/**finalize_spectra**()

Called from *magic*

subroutine spectra/**spectrum**(*n_spec*, *time*, *l_avg*, *n_time_ave*, *l_stop_time*, *time_passed*, *time_norm*, *w*, *dw*, *z*, *b*, *db*, *aj*, *b_ic*, *db_ic*, *aj_ic*)

Calculates magnetic or kinetic energy spectra

Parameters

- **n_spec** *[integer,in]* :: number of spectrum/call, file
- **time** *[real,in]*
- **l_avg** *[logical,in]*
- **n_time_ave** *[integer,in]*
- **l_stop_time** *[logical,in]*
- **time_passed** *[real,in]*
- **time_norm** *[real,in]*
- **w** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **dw** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **z** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **b** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]*
- **db** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]*
- **aj** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]*
- **b_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,in]*
- **db_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,in]*
- **aj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,in]*

Called from *fields_average()*, *output()*

Call to *cc2real()*, *rint_r()*, *cc22real()*, *rintic()*

subroutine spectra/**spectrum_temp**(*n_spec*, *time*, *l_avg*, *n_time_ave*, *l_stop_time*, *time_passed*, *time_norm*, *s*, *ds*)

Computes temperature spectra

Parameters

- **n_spec** *[integer,in]*
- **time** *[real,in]*
- **l_avg** *[logical,in]*
- **n_time_ave** *[integer,in]*
- **l_stop_time** *[logical,in]*
- **time_passed** *[real,in]*
- **time_norm** *[real,in]*
- **s** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **ds** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*

Called from *fields_average()*, *output()*

Call to *cc2real()*, *rint_r()*

subroutine spectra/**get_amplitude**(*time*, *w*, *dw*, *z*, *b*, *db*, *aj*)

Parameters

- **time** *[real,in]*
- **w** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **dw** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **z** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **b** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]*
- **db** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]*
- **aj** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]*

Called from *output()*

Call to *cc2real()*, *rint_r()*

10.18 IO: graphic files, movie files, coeff files and potential files

10.18.1 out_graph_file.f90

Description

This module contains the subroutines that store the 3-D graphic files.

Quick access

Variables *graph_mpi_fh*, *info*, *n_fields*, *n_graph*, *n_graph_file*, *size_of_header*

Routines *close_graph_file()*, *graphout()*, *graphout_header()*, *graphout_ic()*, *graphout_mpi()*, *graphout_mpi_header()*, *open_graph_file()*

Needed modules

- *parallel_mod*: This module contains the blocking information
- *precision_mod*: This module controls the precision used in MagIC
- *constants* (*one()*): module containing constants and parameters used in the code.
- *truncation* (*lm_maxmag()*, *n_r_maxmag()*, *n_r_ic_maxmag()*, *lm_max()*, *n_theta_max()*, *n_phi_tot()*, *n_r_max()*, *l_max()*, *minc()*, *n_phi_max()*, *n_r_ic_max()*, *l_axi()*, *nlat_padded()*): This module defines the grid points and the truncation
- *radial_functions* (*r_cmb()*, *orho1()*, *or1()*, *or2()*, *r()*, *r_icb()*, *r_ic()*, *o_r_ic()*, *o_r_ic2()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *radial_data* (*nrstart()*, *n_r_cmb()*): This module defines the MPI decomposition in the radial direction.

- *physical_parameters* (*ra()*, *ek()*, *pr()*, *prmag()*, *radratio()*, *sigma_ratio()*, *raxi()*, *sc()*, *stef()*): Module containing the physical parameters
- *num_param* (*vscale()*): Module containing numerical and control parameters
- *horizontal_data* (*theta_ord()*, *o_sin_theta()*, *n_theta_cal2ord()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *logic* (*l_mag()*, *l_cond_ic()*, *l_pressgraph()*, *l_chemical_conv()*, *l_heat()*, *l_save_out()*, *l_phase_field()*): Module containing the logicals that control the run
- *output_data* (*runid()*, *n_log_file()*, *log_file()*, *tag()*): This module contains the parameters for output control
- *sht* (*torpol_to_spat_ic()*)

Variables

- graphout_mod/**graph_mpi_fh** *[integer,private]*
- graphout_mod/**info** *[integer,private]*
- graphout_mod/**n_fields** *[integer,private]*
- graphout_mod/**n_graph** *[integer,private/optional/default=0]*
- graphout_mod/**n_graph_file** *[integer,public]*
- graphout_mod/**size_of_header** *[integer,private]*

Subroutines and functions

subroutine graphout_mod/**open_graph_file**(*n_time_step*, *timescaled*)

Parameters

- **n_time_step** *[integer,in]*
- **timescaled** *[real,in]*

Called from *step_time()*

Call to *mpiio_setup()*

subroutine graphout_mod/**close_graph_file**()

Called from *step_time()*

subroutine graphout_mod/**graphout**(*n_r*, *vr*, *vt*, *vp*, *br*, *bt*, *bp*, *sr*, *prer*, *xir*, *phir*)

Output of components of velocity, magnetic field vector, entropy and composition for graphic outputs.

Parameters

- **n_r** *[integer,in]* :: radial grod point no.
- **vr** (,) *[real,in]*
- **vt** (,) *[real,in]*
- **vp** (,) *[real,in]*

- **br** (,) *[real,in]*
- **bt** (,) *[real,in]*
- **bp** (,) *[real,in]*
- **sr** (,) *[real,in]*
- **prer** (,) *[real,in]*
- **xir** (,) *[real,in]*
- **phir** (,) *[real,in]*

Called from *fields_average()*

subroutine graphout_mod/**graphout_header**(*time*)

Parameters **time** *[real,in]*

Called from *fields_average()*

subroutine graphout_mod/**graphout_mpi**(*n_r*, *vr*, *vt*, *vp*, *br*, *bt*, *bp*, *sr*, *prer*, *xir*, *phir*)

MPI version of the graphOut subroutine (use of MPI_IO)

Parameters

- **n_r** *[integer,in]* :: radial grid point no.
- **vr** (,) *[real,in]*
- **vt** (,) *[real,in]*
- **vp** (,) *[real,in]*
- **br** (,) *[real,in]*
- **bt** (,) *[real,in]*
- **bp** (,) *[real,in]*
- **sr** (,) *[real,in]*
- **prer** (,) *[real,in]*
- **xir** (,) *[real,in]*
- **phir** (,) *[real,in]*

Called from *radialloop()*

Call to *write_one_field()*

subroutine graphout_mod/**graphout_mpi_header**(*time*)

Writes the header of the G file (MPI version)

Parameters **time** *[real,in]*

Called from *radialloop()*

subroutine graphout_mod/**write_one_field**(*dummy*, *graph_mpi_fh*, *n_phis*, *n_thetas*)

Parameters

- **dummy** (,) *[real,in]*
- **graph_mpi_fh** *[integer,in]*
- **n_phis** *[integer,in]*
- **n_thetas** *[integer,in]*

subroutine graphout_mod/**graphout_ic**(*b_ic*, *db_ic*, *aj_ic*, *bicb*[, *l_avg*])

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** (,) *[complex,in]*
- **db_ic** (,) *[complex,in]*
- **aj_ic** (,) *[complex,in]*
- **bicb** (*) *[complex,in]*
- **l_avg** *[logical,in,]*

Called from *fields_average()*, *output()*

Call to *torpol_to_spat_ic()*, *write_one_field()*

10.18.2 movie.f90

Quick access

Variables *frames*, *licfield*, *lstoremov*, *movie*, *movie_const*, *movie_file*, *moviedipcolat*, *moviediplon*, *moviedipstrength*, *moviedipstrengthgeo*, *n_frame_work*, *n_md*, *n_movie_const*, *n_movie_field_start*, *n_movie_field_stop*, *n_movie_field_type*, *n_movie_fields*, *n_movie_fields_ic*, *n_movie_fields_max*, *n_movie_file*, *n_movie_surface*, *n_movie_type*, *n_movies*, *n_movies_max*

Routines *finalize_movie_data()*, *get_movie_type()*, *initialize_movie_data()*, *movie_gather_frames_to_rank0()*

Needed modules

- *parallel_mod*: This module contains the blocking information
- *precision_mod*: This module controls the precision used in MagIC
- *truncation* (*n_r_max()*, *n_theta_max()*, *n_phi_max()*, *ldtbmem()*, *minc()*, *n_r_ic_max()*, *lmoviemem()*, *n_r_tot()*): This module defines the grid points and the truncation
- *logic* (*l_store_frame()*, *l_save_out()*, *l_movie()*, *l_movie_oc()*, *l_movie_ic()*, *l_htmovie()*, *l_dtbmovie()*): Module containing the logicals that control the run
- *radial_data* (*nrstart()*, *nrstop()*, *n_r_icb()*, *n_r_cmb()*, *radial_balance()*): This module defines the MPI decomposition in the radial direction.

- *radial_functions* (*r_cmb()*, *r_icb()*, *r()*, *r_ic()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *horizontal_data* (*theta()*, *phi()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *output_data* (*n_log_file()*, *log_file()*, *tag()*): This module contains the parameters for output control
- *charmanip* (*capitalize()*, *delete_string()*, *str2dble()*, *length_to_blank()*, *dble2str()*): This module contains several useful routines to manipule character strings
- *useful* (*logwrite()*, *abortrun()*): This module contains several useful routines.
- *constants* (*pi()*, *one()*): module containing constants and parameters used in the code.
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC

Variables

- movie_data/**frames** (*) *[real,allocatable/public]*
- movie_data/**licfield** (30) *[logical,public]*
- movie_data/**lstoremov** (30) *[logical,public]*
- movie_data/**movie** (30) *[character,public]*
  Only for input
- movie_data/**movie_const** (30) *[real,public]*
- movie_data/**movie_file** (30) *[character,public]*
- movie_data/**moviedipcolat** *[real,public]*
- movie_data/**moviediplon** *[real,public]*
- movie_data/**moviedipstrength** *[real,public]*
- movie_data/**moviedipstrengthgeo** *[real,public]*
- movie_data/**n_frame_work** *[integer,public]*
- movie_data/**n_md** *[integer,public]*
- movie_data/**n_movie_const** (30) *[integer,public]*
- movie_data/**n_movie_field_start** (6,30) *[integer,public]*
- movie_data/**n_movie_field_stop** (6,30) *[integer,public]*
- movie_data/**n_movie_field_type** (6,30) *[integer,public]*
- movie_data/**n_movie_fields** (30) *[integer,public]*
- movie_data/**n_movie_fields_ic** (30) *[integer,public]*
- movie_data/**n_movie_fields_max** *[integer,parameter=6]*
  Max no. of fields per movie
- movie_data/**n_movie_file** (30) *[integer,public]*
- movie_data/**n_movie_surface** (30) *[integer,public]*
- movie_data/**n_movie_type** (30) *[integer,public]*
- movie_data/**n_movies** *[integer,public]*

- movie_data/**n_movies_max** *[integer,parameter=30]*
  Max no. of different movies

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()*

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
- C[omposition] : sic
- AX[ISYMMETRIC] C or AC : axisymmetric C field for phi=constant
- FL Pro : axisymmetric field line stretching
- FL Adv : axisymmetric field line advection
- FL Dif : axisymmetric field line diffusion
- AB Pro : axisymmetric (tor.) Bphi production
- AB Dif : axisymmetric (tor.) Bphi diffusion
- Br Pro : Br production
- Br Adv : Br advection
- Br Dif : Br diffusion
- Jr : Jr production
- Jr Pro : Jr production + omega effects
- Jr Adv : Jr advection
- Jr Dif : Jr diffusion
- Bz Pol : poloidal Bz
- Bz Pol Pro : poloidal Bz production
- Bz Pol Adv : poloidal Bz advection
- Bz Pol Dif : poloidal Bz diffusion
- Jz Tor : poloidal Jz
- Jz Tor Pro : poloidal Jz production
- Jz Tor Adv : poloidal Jz advection
- Jz Tor Dif : poloidal Jz diffusion
- Bp Tor : toriodal Bphi
- Bp Tor Pro : toriodal Bphi production
- Bp Tor Adv : toriodal Bphi advection
- Bp Tor Dif : toriodal Bphi diffusion
- HEL[ICITY] : sic
- AX[ISYMMETRIC HELICITY] or AHEL : axisymmetric helicity
- Bt Tor : toroidal Btheta
- Pot Tor : toroidal Potential
- Pol Fieldlines : toroidal Potential
- Br Shear : azimuthal Shear of Br
- Lorentz[force] : Lorentz force (only phi component)

- Br Inv : Inverse field apperance at CMB

2. A second information that identifies the coordinate to be kept constant (surface). E.g. r=number for surface r=constant with number given in units of the total core radius or theta/phi=number with number given in degrees Four keywords are also possible:
   - CMB : core mantle boundary
   - EQ[UATOR] : equatorial plane
   - SUR[FACE] : Earth surface (only magnetic field)
   - 3[D] : 3D field throughout the OC [and IC for B]

On output the necessary information is coded into integers and is used in this form by further subroutines:

- n_movies = total number of movies
- n_type(n_movie) = movie type:
  - = 1 : Radial magnetic field
  - = 2 : Theta component of magnetic field
  - = 3 : Azimuthal magnetic field
  - = 4 : Horizontal magnetic field
  - = 5 : Total magnetic field (all compnents)
  - = 8 : Axisymmetric azimuthal magnetic field (phi=constant)
  - = 9 : 3d magnetic field
  - = 11 : Radial velocity field
  - = 12 : Theta component of velocity field
  - = 13 : Azimuthal velocity field
  - = 14 : Horizontal velocity field
  - = 15 : Total velocity field (all compnents)
  - = 17 : Scalar field whose contours are the stream lines of the axisymm. poloidal velocity field (phi=constant)
  - = 18 : Axisymmetric azimuthal velocity field (phi=constant)
  - = 19 : 3d velocity field
  - = 20 : z component of vorticity
  - = 21 : Temperature field
  - = 22 : radial conv. heat transport
  - = 23 : helicity
  - = 24 : axisymmetric helicity
  - = 25 : phi component of vorticity
  - = 26 : radial component of vorticity
  - = 28 : axisymmetric Temperature field for phi=const.
  - = 29 : 3d temperature field

- = 30 : Scalar field whose contours are the fieldlines of the axisymm. poloidal magnetic field (phi=constant)
- = 31 : field line production
- = 32 : field line advection
- = 33 : field line diffusion
- = 40 : Axisymmetric azimuthal magnetic field (phi=constant)
- = 41 : Axis. Bphi production + omega eff.
- = 42 : Axis. Bphi advection
- = 43 : Axis. Bphi diffusion
- = 44 : Axis. Bphi str.,dyn.,omega,diff.
- = 50 : Bz
- = 51 : Bz production
- = 52 : Bz advection
- = 53 : Bz diffusion
- = 60 : toroidal Bphi
- = 61 : toroidal Bphi production + omega eff.
- = 62 : toroidal Bphi advection
- = 63 : toroidal Bphi diffusion
- = 71 : Br production
- = 72 : Br advection
- = 73 : Br diffusion
- = 80 : Jr
- = 81 : Jr production
- = 82 : Jr advection
- = 83 : Jr diffusion
- = 90 : poloidal Jz pol.
- = 91 : poloidal Jz pol. production
- = 92 : poloidal Jz advection
- = 93 : poloidal Jz diffusion
- = 94 : z component of velovity
- = 95 : toroidal Btheta
- = 96 : toroidal Potential
- = 97 : Function for Poloidal Fieldlines
- = 98 : azimuthal shear of Br
- = 99 : phi component of Lorentz force
- =101 : Stress fields

  - =102 : Force fields
  - =103 : Br Inverse appearence at CMB
  - =110 : radial heat flow
  - =111 : Vz and Vorz north/south correlation
  - =112 : axisymm dtB tersm for Br and Bp
  - =114 : Cylindrically radial magnetic field
  - =115 : Composition field
  - =116 : axisymmetric Vs
  - =117 : axisymmetric Composition field
  - =118 : axisymmetric phase field
  - =121 : phase field for phi=const.
- 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
  - =109: composition field
  - =110: axisymm. composition
  - =111: axisymm. phase
  - =112: phase field

- 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()*, *abortrun()*, *str2dble()*, *dble2str()*, *length_to_blank()*

subroutine movie_data/**movie_gather_frames_to_rank0**()

MPI communicators for movie files

Called from *output()*

Call to *abortrun()*

10.18.3 out_movie_file.f90

Quick access

Routines *get_b_surface()*, *get_fl()*, *get_sl()*, *store_fields_3d()*, *store_fields_p()*, *store_fields_r()*, *store_fields_sur()*, *store_fields_t()*, *store_movie_frame()*, *write_movie_frame()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod* (rank()): This module contains the blocking information
- *communications* (*gt_oc()*, *gather_all_from_lo_to_rank0()*): This module contains the different MPI communicators used in MagIC.
- *truncation* (*n_phi_max()*, *n_theta_max()*, *minc()*, *lm_max()*, *l_max()*, *n_m_max()*, *lm_maxmag()*, *n_r_maxmag()*, *n_r_ic_maxmag()*, *n_r_ic_max()*, *n_r_max()*, *l_axi()*, *nlat_padded()*): This module defines the grid points and the truncation
- *movie_data* (*frames()*, *n_movie_fields()*, *n_movies()*, *n_movie_surface()*, *n_movie_const()*, *n_movie_field_type()*, *n_movie_field_start()*, *n_movie_field_stop()*, *moviedipcolat()*, *moviediplon()*, *moviedipstrength()*, *moviedipstrengthgeo()*, *n_movie_type()*, *lstoremov()*, *n_movie_file()*, *n_movie_fields_ic()*, *movie_file()*, *movie_const()*)
- *radial_data* (*n_r_icb()*, *n_r_cmb()*): This module defines the MPI decomposition in the radial direction.
- *radial_functions* (*orho1()*, *orho2()*, *or1()*, *or2()*, *or3()*, *or4()*, *beta()*, *r_surface()*, *r_cmb()*, *r()*, *r_ic()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *physical_parameters* (*lffac()*, *radratio()*, *ra()*, *ek()*, *pr()*, *prmag()*): Module containing the physical parameters
- *num_param* (*vscale()*, *tscale()*): Module containing numerical and control parameters
- *blocking* (*lm2l()*, *lm2()*, *llmmag()*, *ulmmag()*): Module containing blocking information
- *horizontal_data* (*o_sin_theta()*, *sintheta()*, *costheta()*, *n_theta_cal2ord()*, *o_sin_theta_e2()*, *osn1()*, *phi()*, *theta_ord()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *fields* (*w_rloc()*, *b_rloc()*, *b_ic()*, *bicb()*): This module contains all the fields used in MagIC in the hybrid (LM,r) space as well as their radial derivatives. It defines both the LM-distributed arrays and the R-distributed arrays....
- *sht* (*torpol_to_spat()*, *toraxi_to_spat()*)
- *logic* (*l_save_out()*, *l_cond_ic()*, *l_mag()*): Module containing the logicals that control the run
- *constants* (*zero()*, *one()*, *two()*): module containing constants and parameters used in the code.
- *out_dtb_frame* (*write_dtb_frame()*)
- *output_data* (*runid()*): This module contains the parameters for output control
- *useful* (*abortrun()*): This module contains several useful routines.

Variables

Subroutines and functions

subroutine out_movie/**store_movie_frame**(*n_r*, *vr*, *vt*, *vp*, *br*, *bt*, *bp*, *sr*, *drsr*, *xir*, *phir*, *dvrdp*, *dvpdr*, *dvtdr*, *dvrdt*, *cvr*, *cbr*, *cbt*)

Controls output of movie frames. Usually called from radialLoop.

Parameters

- **n_r** *[integer,in]* :: radial grid point no.
- **vr** (,) *[real,in]*
- **vt** (,) *[real,in]*
- **vp** (,) *[real,in]*
- **br** (,) *[real,in]*
- **bt** (,) *[real,in]*
- **bp** (,) *[real,in]*
- **sr** (,) *[real,in]*
- **drsr** (,) *[real,in]*
- **xir** (,) *[real,in]*
- **phir** (,) *[real,in]*
- **dvrdp** (,) *[real,in]*
- **dvpdr** (,) *[real,in]*
- **dvtdr** (,) *[real,in]*
- **dvrdt** (,) *[real,in]*
- **cvr** (,) *[real,in]*
- **cbr** (,) *[real,in]*
- **cbt** (,) *[real,in]*

Called from *radialloop()*

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_lmloc*, *db_lmloc*, *aj_lmloc*, *dj_lmloc*, *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_lmloc** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]*

- **db_lmloc** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]*
- **aj_lmloc** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]*
- **dj_lmloc** (1 - *llmmag* + *ulmmag*,*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 *gather_all_from_lo_to_rank0()*, *abortrun()*, *write_dtb_frame()*

subroutine out_movie/**store_fields_sur**(*n_store_last*, *n_field_type*, *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
- **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*, *xir*, *phir*, *dvrdp*, *dvpdr*, *dvtdr*, *dvrdt*, *cvr*, *n_r*, *n_store_last*, *n_field_type*)

Purpose of this subroutine is to store movie frames for surfaces r=const. into array frame(,)

Parameters

- **vr** (,) *[real,in]*
- **vt** (,) *[real,in]*
- **vp** (,) *[real,in]*
- **br** (,) *[real,in]*
- **bt** (,) *[real,in]*
- **bp** (,) *[real,in]*
- **sr** (,) *[real,in]*
- **drsr** (,) *[real,in]*
- **xir** (,) *[real,in]*
- **phir** (,) *[real,in]*
- **dvrdp** (,) *[real,in]*

- **dvpdr** (,) *[real,in]*
- **dvtdr** (,) *[real,in]*
- **dvrdt** (,) *[real,in]*
- **cvr** (,) *[real,in]*
- **n_r** *[integer,in]*
- **n_store_last** *[integer,in]* :: Start position in frame(*)-1
- **n_field_type** *[integer,in]* :: Defines field type

Called from *store_movie_frame()*

subroutine out_movie/**store_fields_p**(*vr*, *vt*, *vp*, *br*, *bp*, *bt*, *sr*, *drsr*, *xir*, *phir*, *dvrdp*, *dvpdr*, *dvtdr*, *dvrdt*, *cvr*, *cbr*, *cbt*, *n_r*, *n_store_last*, *n_field_type*, *n_phi_const*, *n_field_size*)

Purpose of this subroutine is to store movie frames for surfaces phi=const. into array frames(,)

Parameters

- **vr** (,) *[real,in]*
- **vt** (,) *[real,in]*
- **vp** (,) *[real,in]*
- **br** (,) *[real,in]*
- **bp** (,) *[real,in]*
- **bt** (,) *[real,in]*
- **sr** (,) *[real,in]*
- **drsr** (,) *[real,in]*
- **xir** (,) *[real,in]*
- **phir** (,) *[real,in]*
- **dvrdp** (,) *[real,in]*
- **dvpdr** (,) *[real,in]*
- **dvtdr** (,) *[real,in]*
- **dvrdt** (,) *[real,in]*
- **cvr** (,) *[real,in]*
- **cbr** (,) *[real,in]*
- **cbt** (,) *[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

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*, *xir*, *phir*, *dvrdp*, *dvpdr*, *dvtdr*, *dvrdt*, *cvr*, *cbt*, *n_r*, *n_store_last*, *n_field_type*, *n_theta*)

Purpose of this subroutine is to store movie frames for surfaces r=const. into array frame(,)

Parameters

- **vr** (,) *[real,in]*
- **vt** (,) *[real,in]*
- **vp** (,) *[real,in]*
- **br** (,) *[real,in]*
- **bt** (,) *[real,in]*
- **bp** (,) *[real,in]*
- **sr** (,) *[real,in]*
- **drsr** (,) *[real,in]*
- **xir** (,) *[real,in]*
- **phir** (,) *[real,in]*
- **dvrdp** (,) *[real,in]*
- **dvpdr** (,) *[real,in]*
- **dvtdr** (,) *[real,in]*
- **dvrdt** (,) *[real,in]*
- **cvr** (,) *[real,in]*
- **cbt** (,) *[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** *[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*, *xir*, *phir*, *dvrdp*, *dvpdr*, *dvtdr*, *dvrdt*, *cvr*, *cbr*, *cbt*, *n_r*, *n_store_last*, *n_field_type*)

Purpose of this subroutine is to store movie frames for surfaces r=const. into array frame(,)

Parameters

- **vr** (,) *[real,in]*
- **vt** (,) *[real,in]*
- **vp** (,) *[real,in]*
- **br** (,) *[real,in]*
- **bt** (,) *[real,in]*

- **bp** (,) *[real,in]*
- **sr** (,) *[real,in]*
- **drsr** (,) *[real,in]*
- **xir** (,) *[real,in]*
- **phir** (,) *[real,in]*
- **dvrdp** (,) *[real,in]*
- **dvpdr** (,) *[real,in]*
- **dvtdr** (,) *[real,in]*
- **dvrdt** (,) *[real,in]*
- **cvr** (,) *[real,in]*
- **cbr** (,) *[real,in]*
- **cbt** (,) *[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

Called from *store_movie_frame()*

subroutine out_movie/**get_sl**(*sl*, *n_r*)

Return field sl whose contourlines are the stream lines of the axisymmetric poloidal velocity field.

$$s(r,\theta) = \frac{1}{r}\frac{\partial}{\partial\theta}u(r,\theta,m=0)$$

Parameters

- **sl** (*) *[real,out]* :: Field for field lines
- **n_r** *[integer,in]* :: No. of radial grid point

Called from *store_fields_p()*

Call to *toraxi_to_spat()*

subroutine out_movie/**get_fl**(*fl*, *n_r*, *l_ic*)

Return field fl whose contourlines are the fields lines of the axisymmetric poloidal mangetic field.

$$f(r,\theta) = \frac{1}{r}\frac{\partial}{\partial\theta}b(r,\theta,m=0)$$

This routine is called for l_ic=.true. only from rank 0 with full field b_ic in standard lm ordering available. The case l_ic=.false. is called from all ranks and uses b_Rloc.

Parameters

- **fl** (*) *[real,out]* :: Field for field lines
- **n_r** *[integer,in]* :: No. of radial grid point
- **l_ic** *[logical,in]* :: =true if inner core field

Called from *store_fields_p()*, *store_movie_frame_ic()*

Call to *toraxi_to_spat()*

subroutine out_movie/**get_b_surface**(*b_r*, *b_t*, *b_p*, *bcmb*)

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: $r^2 B_r$, $r^2 \sin\theta B_\theta$, $r^2 \sin\theta B_\phi$

Parameters

- **b_r** (,) *[real,out]* :: Radial magnetic field in (theta,phi)-space
- **b_t** (,) *[real,out]* :: Latitudinal magnetic field
- **b_p** (,) *[real,out]* :: Azimuthal magnetic field.
- **bcmb** (*) *[complex,in]*

Called from *store_fields_sur()*

Call to *torpol_to_spat()*

10.18.4 store_movie_IC.f90

Quick access

Routines *store_movie_frame_ic()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *truncation* (*minc()*, *lm_maxmag()*, *n_r_maxmag()*, *n_r_ic_maxmag()*, *n_phi_max()*, *lm_max()*, *n_r_ic_max()*, *l_max()*, *n_theta_max()*, *l_axi()*, *nlat_padded()*): This module defines the grid points and the truncation
- *radial_data* (*n_r_icb()*): This module defines the MPI decomposition in the radial direction.
- *radial_functions* (*r_ic()*, *r_icb()*, *o_r_ic2()*, *o_r_ic()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *physical_parameters* (*lffac()*): Module containing the physical parameters
- *horizontal_data* (*n_theta_cal2ord()*, *o_sin_theta()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *logic* (*l_cond_ic()*): Module containing the logicals that control the run
- *movie_data* (*frames()*, *n_movie_field_stop()*, *n_movie_field_start()*, *n_movie_type()*, *n_movie_const()*, *n_movie_fields_ic()*, *n_movie_surface()*, *n_movies()*, *n_movie_field_type()*, *n_movie_fields()*)
- *out_movie* (*get_fl()*)
- *constants* (*one()*): module containing constants and parameters used in the code.
- *sht* (*torpol_to_spat_ic()*, *torpol_to_curl_spat_ic()*)

Variables

Subroutines and functions

subroutine out_movie_ic/**store_movie_frame_ic**(*bicb*, *b_ic*, *db_ic*, *ddb_ic*, *aj_ic*, *dj_ic*)

Controls storage of IC magnetic field in movie frame.

Parameters

- **bicb** (*lm_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 *torpol_to_spat_ic()*, *torpol_to_curl_spat_ic()*, *get_fl()*

10.18.5 out_coeff.f90

Description

This module contains the subroutines that calculate the Bcmb files, the [B|V|T]_coeff_r files and the [B|V|T]_lmr files

Quick access

Variables *b_r_file*, *n_b_r_file*, *n_b_r_sets*, *n_t_r_file*, *n_t_r_sets*, *n_v_r_file*, *n_v_r_sets*, *n_xi_r_file*, *n_xi_r_sets*, *t_r_file*, *v_r_file*, *xi_r_file*

Routines *finalize_coeff()*, *initialize_coeff()*, *write_bcmb()*, *write_coeff_r()*, *write_coeffs()*, *write_pot()*, *write_pot_mpi()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod*: This module contains the blocking information
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *logic* (*l_r_field()*, *l_cmb_field()*, *l_save_out()*, *l_average()*, *l_cond_ic()*, *l_r_fieldt()*, *l_r_fieldxi()*, *l_mag()*): Module containing the logicals that control the run
- *radial_functions* (*r()*, *rho0()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *radial_data* (*nrstart()*, *nrstop()*): This module defines the MPI decomposition in the radial direction.
- *physical_parameters* (*ra()*, *ek()*, *pr()*, *prmag()*, *radratio()*, *sigma_ratio()*, *raxi()*, *sc()*): Module containing the physical parameters

- *num_param* (*tscale()*): Module containing numerical and control parameters
- *blocking* (*lm2()*, *llm()*, *ulm()*): Module containing blocking information
- *truncation* (*lm_max()*, *l_max()*, *minc()*, *n_r_max()*, *n_r_ic_max()*): This module defines the grid points and the truncation
- *communications* (*gather_from_lo_to_rank0()*, *gather_all_from_lo_to_rank0()*, *gt_ic()*, *gt_oc()*): This module contains the different MPI communicators used in MagIC.
- *output_data* (*tag()*, *n_coeff_r()*, *n_r_array()*, *n_r_step()*, *l_max_r()*, *n_coeff_r_max()*): This module contains the parameters for output control
- *constants* (*two()*, *half()*): module containing constants and parameters used in the code.

Variables

- out_coeff/**b_r_file** (*) *[character,private/allocatable]*
- out_coeff/**filehandle** *[integer,private]*
- out_coeff/**n_b_r_file** (*) *[integer,private/allocatable]*
- out_coeff/**n_b_r_sets** (*) *[integer,private/allocatable]*
- out_coeff/**n_t_r_file** (*) *[integer,private/allocatable]*
- out_coeff/**n_t_r_sets** (*) *[integer,private/allocatable]*
- out_coeff/**n_v_r_file** (*) *[integer,private/allocatable]*
- out_coeff/**n_v_r_sets** (*) *[integer,private/allocatable]*
- out_coeff/**n_xi_r_file** (*) *[integer,private/allocatable]*
- out_coeff/**n_xi_r_sets** (*) *[integer,private/allocatable]*
- out_coeff/**t_r_file** (*) *[character,private/allocatable]*
- out_coeff/**v_r_file** (*) *[character,private/allocatable]*
- out_coeff/**xi_r_file** (*) *[character,private/allocatable]*

Subroutines and functions

subroutine out_coeff/**initialize_coeff**()

Called from *initialize_output()*

subroutine out_coeff/**finalize_coeff**()

Called from *finalize_output()*

subroutine out_coeff/**write_bcmb**(*time*, *b_lmloc*, *l_max_cmb*, *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 header of cmb_file. Each further set contains:

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_lmloc** (1 - *llm* + *ulm*) *[complex,in]* :: Poloidal field potential
- **l_max_cmb** *[integer,inout]* :: Max degree of output
- **n_cmb_sets** *[integer,inout]* :: Total no. of cmb sets,
- **cmb_file** *[character,in]* :: Name of output file
- **n_cmb_file** *[integer,inout]* :: Output unit for $cmb_file

Called from *fields_average()*, *output()*

Call to *gather_from_lo_to_rank0()*

subroutine out_coeff/**write_coeffs**(*w_lmloc*, *dw_lmloc*, *ddw_lmloc*, *z_lmloc*, *b_lmloc*, *db_lmloc*, *ddb_lmloc*, *aj_lmloc*, *dj_lmloc*, *s_lmloc*, *xi_lmloc*, *timescaled*)

This routine handles the writing of coefficients at a given depth

Parameters

- **w_lmloc** (,) *[complex,in]*
- **dw_lmloc** (,) *[complex,in]*
- **ddw_lmloc** (,) *[complex,in]*
- **z_lmloc** (,) *[complex,in]*
- **b_lmloc** (,) *[complex,in]*
- **db_lmloc** (,) *[complex,in]*
- **ddb_lmloc** (,) *[complex,in]*
- **aj_lmloc** (,) *[complex,in]*
- **dj_lmloc** (,) *[complex,in]*
- **s_lmloc** (,) *[complex,in]*
- **xi_lmloc** (,) *[complex,in]*
- **timescaled** *[real,in]*

Called from *output()*

Call to *write_coeff_r()*

subroutine out_coeff/**write_coeff_r**(*time*, *w_lmloc*, *dw_lmloc*, *ddw_lmloc*, *z_lmloc*, *r*, *l_max_r*, *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:

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_lmloc** (1 - *llm* + *ulm*) *[complex,in]* :: Poloidal field potential
- **dw_lmloc** (1 - *llm* + *ulm*) *[complex,in]* :: dr of Poloidal field potential
- **ddw_lmloc** (1 - *llm* + *ulm*) *[complex,in]* :: dr^2 of Poloidal field potential
- **z_lmloc** (1 - *llm* + *ulm*) *[complex,in]* :: Toroidal field potential
- **r** *[real,in]* :: radius of coeffs
- **l_max_r** *[integer,inout]* :: Max degree of output
- **n_sets** *[integer,inout]* :: Total no. of cmb sets,
- **file** *[character,in]* :: Name of output file
- **n_file** *[integer,inout]* :: Output unit for $file
- **nvbs** *[integer,in]* :: 1 if flow, 2 if magnetic field and 3 if temperature or composition

Called from *write_coeffs()*

Call to *gather_from_lo_to_rank0()*

subroutine out_coeff/**write_pot_mpi**(*time*, *b*, *aj*, *b_ic*, *aj_ic*, *npotsets*, *root*, *omega_ma*, *omega_ic*)

This routine stores the fields in (lm,r) space using MPI-IO

Parameters

- **time** *[real,in]* :: Time
- **b** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,in]* :: Poloidal potential
- **aj** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,in]* :: Toroidal potential
- **b_ic** (1 - *llm* + *ulm*,*n_r_ic_max*) *[complex,in]*
- **aj_ic** (1 - *llm* + *ulm*,*n_r_ic_max*) *[complex,in]*
- **npotsets** *[integer,in]*
- **root** *[character,in]*
- **omega_ma** *[real,in]*
- **omega_ic** *[real,in]*

Called from *output()*

Call to *mpiio_setup()*, *gather_all_from_lo_to_rank0()*

subroutine out_coeff/**write_pot**(*time*, *b*, *aj*, *b_ic*, *aj_ic*, *npotsets*, *root*, *omega_ma*, *omega_ic*)

This routine stores the fields in spectral and radial space

Parameters

- **time** *[real,in]* :: Output time
- **b** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **aj** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*

- **b_ic** (1 - *llm* + *ulm*,*n_r_ic_max*) *[complex,in]*
- **aj_ic** (1 - *llm* + *ulm*,*n_r_ic_max*) *[complex,in]*
- **npotsets** *[integer,in]*
- **root** *[character,in]* :: File prefix
- **omega_ma** *[real,in]*
- **omega_ic** *[real,in]*

Called from *fields_average()*

Call to *gather_all_from_lo_to_rank0()*

10.18.6 field_average.f90

Description

This module is used when one wants to store time-averaged quantities

Quick access

Variables *aj_ave*, *aj_ave_global*, *aj_ic_ave*, *b_ave*, *b_ave_global*, *b_ic_ave*, *bicb*, *db_ave_global*, *dw_ave_global*, *p_ave*, *p_ave_global*, *phi_ave*, *phi_ave_global*, *s_ave*, *s_ave_global*, *w_ave*, *w_ave_global*, *xi_ave*, *xi_ave_global*, *z_ave*, *z_ave_global*

Routines *fields_average()*, *finalize_fields_average_mod()*, *initialize_fields_average_mod()*

Needed modules

- *truncation*: This module defines the grid points and the truncation
- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *radial_data* (*n_r_cmb()*, *n_r_icb()*): This module defines the MPI decomposition in the radial direction.
- *radial_functions* (*chebt_ic()*, *chebt_ic_even()*, *r()*, *dr_fac_ic()*, `rscheme_oc()`, *l_r()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *blocking* (*lm2()*, *llm()*, *ulm()*, *llmmag()*, *ulmmag()*): Module containing blocking information
- *logic* (*l_mag()*, *l_conv()*, *l_save_out()*, *l_heat()*, *l_cond_ic()*, *l_chemical_conv()*, *l_phase_field()*): Module containing the logicals that control the run
- *kinetic_energy* (*get_e_kin()*)
- *magnetic_energy* (*get_e_mag()*): This module handles the computation and the writing of the diagnostic files related to magnetic energy: e_mag_oc.TAG, e_mag_ic.TAG, dipole.TAG, eMagR.TAG
- *output_data* (*tag()*, *n_log_file()*, *log_file()*, *n_graphs()*, *l_max_cmb()*): This module contains the parameters for output control
- *parallel_mod* (`rank()`): This module contains the blocking information
- *sht* (*torpol_to_spat()*, *scal_to_spat()*)

- *constants* (*zero()*, *vol_oc()*, *vol_ic()*, *one()*): module containing constants and parameters used in the code.
- *communications* (*get_global_sum()*, *gather_from_lo_to_rank0()*, *gather_all_from_lo_to_rank0()*, *gt_oc()*, *gt_ic()*): This module contains the different MPI communicators used in MagIC.
- *out_coeff* (*write_bcmb()*, *write_pot()*): This module contains the subroutines that calculate the Bcmb files, the [B|V|T]_coeff_r files and the [B|V|T]_lmr files
- *spectra* (*spectrum()*, *spectrum_temp()*)
- *graphout_mod* (*graphout()*, *graphout_ic()*, *n_graph_file()*, *graphout_header()*): This module contains the subroutines that store the 3-D graphic files.
- *radial_der_even* (*get_drns_even()*, *get_ddrns_even()*)
- *radial_der* (*get_dr()*): Radial derivatives functions
- *fieldslast* (*dwdt()*, *dpdt()*, *dzdt()*, *dsdt()*, *dxidt()*, *dbdt()*, *djdt()*, *dbdt_ic()*, *djdt_ic()*, *domega_ma_dt()*, *domega_ic_dt()*, *dphidt()*, *lorentz_torque_ic_dt()*, *lorentz_torque_ma_dt()*): This module contains all the work arrays of the previous time-steps needed to time advance the code. They are needed in the time-stepping scheme....
- *storecheckpoints* (*store()*): This module contains several subroutines that can be used to store the checkpoint_#.tag files
- *time_schemes* (`type_tscheme()`): This module defines an abstract class type_tscheme which is employed for the time advance of the code.

Variables

- fields_average_mod/**aj_ave** (*,*) *[complex,private/allocatable]*
- fields_average_mod/**aj_ave_global** (*) *[complex,private/allocatable]*
- fields_average_mod/**aj_ic_ave** (*,*) *[complex,private/allocatable]*
- fields_average_mod/**b_ave** (*,*) *[complex,private/allocatable]*
- fields_average_mod/**b_ave_global** (*) *[complex,private/allocatable]*
- fields_average_mod/**b_ic_ave** (*,*) *[complex,private/allocatable]*
- fields_average_mod/**bicb** (*) *[complex,private/allocatable]*
- fields_average_mod/**db_ave_global** (*) *[complex,private/allocatable]*
- fields_average_mod/**dw_ave_global** (*) *[complex,private/allocatable]*
- fields_average_mod/**p_ave** (*,*) *[complex,private/allocatable]*
- fields_average_mod/**p_ave_global** (*) *[complex,private/allocatable]*
- fields_average_mod/**phi_ave** (*,*) *[complex,private/allocatable]*
- fields_average_mod/**phi_ave_global** (*) *[complex,private/allocatable]*
- fields_average_mod/**s_ave** (*,*) *[complex,private/allocatable]*
- fields_average_mod/**s_ave_global** (*) *[complex,private/allocatable]*
- fields_average_mod/**w_ave** (*,*) *[complex,private/allocatable]*
- fields_average_mod/**w_ave_global** (*) *[complex,private/allocatable]*

- fields_average_mod/**xi_ave** (*,*) *[complex,private/allocatable]*
- fields_average_mod/**xi_ave_global** (*) *[complex,private/allocatable]*
- fields_average_mod/**z_ave** (*,*) *[complex,private/allocatable]*
- fields_average_mod/**z_ave_global** (*) *[complex,private/allocatable]*

Subroutines and functions

subroutine fields_average_mod/**initialize_fields_average_mod**()

Called from *magic*

subroutine fields_average_mod/**finalize_fields_average_mod**()

Called from *magic*

subroutine fields_average_mod/**fields_average**(*simtime*, *tscheme*, *nave*, *l_stop_time*, *time_passed*, *time_norm*, *omega_ic*, *omega_ma*, *w*, *z*, *p*, *s*, *xi*, *phi*, *b*, *aj*, *b_ic*, *aj_ic*)

This subroutine averages fields b and v over time.

Parameters

- **simtime** *[real,in]*
- **tscheme** *[real]*
- **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** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **z** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **p** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **s** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **xi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **phi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **b** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]*
- **aj** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]*
- **b_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,in]*
- **aj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,in]*

Called from *output()*

Call to *get_ddrns_even()*, *get_drns_even()*, *spectrum()*, *spectrum_temp()*, *get_e_kin()*, *get_e_mag()*, *graphout_header()*, *gather_from_lo_to_rank0()*, *torpol_to_spat()*, *scal_to_spat()*, *graphout()*, *gather_all_from_lo_to_rank0()*, *graphout_ic()*, *write_bcmb()*, *write_pot()*, *store()*

10.19 IO: RMS force balance, torsional oscillations, misc

10.19.1 RMS.f90

Description

This module contains the calculation of the RMS force balance and induction terms.

Quick access

Variables *adv2hint*, *advp2*, *advp2lm*, *advrmsl*, *advrmslnr*, *advt2*, *advt2lm*, *arc2hint*, *arcmag2hint*, *arcmagrmsl*, *arcmagrmslnr*, *arcrmsl*, *arcrmslnr*, *buo_temp2hint*, *buo_temprmsl*, *buo_temprmslnr*, *buo_xi2hint*, *buo_xirmsl*, *buo_xirmslnr*, *cfp2*, *cfp2lm*, *cft2*, *cft2lm*, *cia2hint*, *ciarmsl*, *ciarmslnr*, *clf2hint*, *clfrmsl*, *clfrmslnr*, *cor2hint*, *corrmsl*, *corrmslnr*, *difpol2hint*, *difpollmr*, *difrmsl*, *difrmslnr*, *diftor2hint*, *dpdpc*, *dpdtc*, *dpkindrc*, *dpkindrlm*, *dr_facc*, *dtbpol2hint*, *dtbpollmr*, *dtbrms_file*, *dtbtor2hint*, *dtvp*, *dtvplm*, *dtvr*, *dtvrlm*, *dtvrms_file*, *dtvt*, *dtvtlm*, *geo2hint*, *geormsl*, *geormslnr*, *iner2hint*, *inerrmsl*, *inerrmslnr*, *lf2hint*, *lfp2*, *lfp2lm*, *lfrlm*, *lfrmsl*, *lfrmslnr*, *lft2*, *lft2lm*, *mag2hint*, *magrmsl*, *magrmslnr*, *n_cheb_maxc*, *n_dtbrms_file*, *n_dtvrms_file*, *n_r_maxc*, *ncut*, *pfp2lm*, *pft2lm*, *plf2hint*, *plfrmsl*, *plfrmslnr*, *pre2hint*, *prermsl*, *prermslnr*, *rc*, *vp_old*, *vr_old*, *vt_old*

Routines *compute_lm_forces()*, *dtbrms()*, *dtvrms()*, *finalize_rms()*, *get_force()*, *get_nl_rms()*, *init_rnb()*, *initialize_rms()*, *transform_to_grid_rms()*, *transform_to_lm_rms()*, *zerorms()*

Needed modules

- *parallel_mod*: This module contains the blocking information
- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *blocking* (*st_map()*, *lo_map()*, *lm2()*, *lm2m()*, *llm()*, *ulm()*, *llmmag()*, *ulmmag()*, *lm2lma()*, *lm2lmp()*, *lm2l()*, *lm2lms()*): Module containing blocking information
- *finite_differences* (`type_fd()`): This module is used to calculate the radial grid when finite differences are requested
- *chebyshev* (`type_cheb_odd()`)
- *radial_scheme* (`type_rscheme()`): This is an abstract type that defines the radial scheme used in MagIC
- *truncation* (*n_r_max()*, *n_cheb_max()*, *n_r_maxmag()*, *lm_max()*, *lm_maxmag()*, *l_max()*, *n_phi_max()*, *n_theta_max()*, *minc()*, *n_r_max_dtb()*, *lm_max_dtb()*, *fd_ratio()*, *fd_stretch()*, *lmp_max()*, *nlat_padded()*): This module defines the grid points and the truncation

- *physical_parameters* (*ra()*, *ek()*, *pr()*, *prmag()*, *radratio()*, *corfac()*, *n_r_lcr()*, *buofac()*, *chemfac()*, *thexpnb()*, *vischeatfac()*): Module containing the physical parameters
- *radial_data* (*nrstop()*, *nrstart()*, *radial_balance()*, *nrstartmag()*, *nrstopmag()*): This module defines the MPI decomposition in the radial direction.
- *radial_functions* (`rscheme_oc()`, *r()*, *r_cmb()*, *r_icb()*, *or1()*, *or2()*, *or3()*, *or4()*, *rho0()*, *rgrav()*, *beta()*, *dlvisc()*, *dbeta()*, *ogrun()*, *alpha0()*, *temp0()*, *visc()*, *l_r()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *logic* (*l_save_out()*, *l_heat()*, *l_chemical_conv()*, *l_conv_nl()*, *l_mag_lf()*, *l_conv()*, *l_corr()*, *l_mag()*, *l_finite_diff()*, *l_newmap()*, *l_2d_rms()*, *l_parallel_solve()*, *l_mag_par_solve()*, *l_adv_curl()*, *l_double_curl()*, *l_anelastic_liquid()*, *l_mag_nl()*, *l_non_rot()*): Module containing the logicals that control the run
- *num_param* (*tscale()*, *alph1()*, *alph2()*): Module containing numerical and control parameters
- *horizontal_data* (*phi()*, *theta_ord()*, *costheta()*, *sintheta()*, *o_sin_theta_e2()*, *cosn_theta_e2()*, *o_sin_theta()*, *dtheta2a()*, *dphi()*, *dtheta2s()*, *dlh()*, *hdif_v()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *constants* (*zero()*, *one()*, *half()*, *four()*, *third()*, *vol_oc()*, *pi()*, *two()*, *three()*): module containing constants and parameters used in the code.
- *integration* (*rint_r()*): Radial integration functions
- *radial_der* (*get_dr()*, *get_dr_rloc()*): Radial derivatives functions
- *output_data* (*rdea()*, *rcut()*, *tag()*, *runid()*): This module contains the parameters for output control
- *cosine_transform_odd*: This module contains the built-in type I discrete Cosine Transforms. This implementation is based on Numerical Recipes and FFTPACK. This only works for `n_r_max-1 = 2**a 3**b 5**c`, with a,b,c integers....
- *rms_helpers* (*hint2dpol()*, *get_poltorrms()*, *hint2dpollm()*, *hintrms()*): This module contains several useful subroutines required to compute RMS diagnostics
- *dtb_mod* (*pdiflm_lmloc()*, *tdiflm_lmloc()*, *pstrlm_lmloc()*, *padvlm_lmloc()*, *tadvlm_lmloc()*, *tstrlm_lmloc()*, *tomelm_lmloc()*): This module contains magnetic field stretching and advection terms plus a separate omega-effect. It is used for movie output....
- *useful* (*abortrun()*): This module contains several useful routines.
- *mean_sd* (`mean_sd_type()`, `mean_sd_2d_type()`): This module contains a small type that simply handles two arrays (mean and SD) This type is used for time-averaged outputs (and their standard deviations).
- *time_schemes* (`type_tscheme()`): This module defines an abstract class type_tscheme which is employed for the time advance of the code.
- *sht* (*spat_to_sphertor()*, *spat_to_qst()*, *scal_to_sh()*, *scal_to_grad_spat()*)

Variables

- rms/**adv2hint** (*,*) *[real,private/allocatable]*
- rms/**advp2** (*,*) *[real,private/allocatable]*
- rms/**advp2lm** (*) *[complex,private/allocatable]*
- rms/**advrmsl** *[mean_sd_type,private]*
- rms/**advrmslnr** *[mean_sd_2d_type,private]*
- rms/**advt2** (*,*) *[real,private/allocatable]*

- rms/**advt2lm** (*) *[complex,private/allocatable]*
- rms/**arc2hint** (*,*) *[real,private/allocatable]*
- rms/**arcmag2hint** (*,*) *[real,private/allocatable]*
- rms/**arcmagrmsl** *[mean_sd_type,private]*
- rms/**arcmagrmslnr** *[mean_sd_2d_type,private]*
- rms/**arcrmsl** *[mean_sd_type,private]*
- rms/**arcrmslnr** *[mean_sd_2d_type,private]*
- rms/**buo_temp2hint** (*,*) *[real,private/allocatable]*
- rms/**buo_temprmsl** *[mean_sd_type,private]*
- rms/**buo_temprmslnr** *[mean_sd_2d_type,private]*
- rms/**buo_xi2hint** (*,*) *[real,private/allocatable]*
- rms/**buo_xirmsl** *[mean_sd_type,private]*
- rms/**buo_xirmslnr** *[mean_sd_2d_type,private]*
- rms/**cfp2** (*,*) *[real,private/allocatable]*
- rms/**cfp2lm** (*) *[complex,private/allocatable]*
- rms/**cft2** (*,*) *[real,private/allocatable]*
- rms/**cft2lm** (*) *[complex,private/allocatable]*
- rms/**cia2hint** (*,*) *[real,private/allocatable]*
- rms/**ciarmsl** *[mean_sd_type,private]*
- rms/**ciarmslnr** *[mean_sd_2d_type,private]*
- rms/**clf2hint** (*,*) *[real,private/allocatable]*
- rms/**clfrmsl** *[mean_sd_type,private]*
- rms/**clfrmslnr** *[mean_sd_2d_type,private]*
- rms/**cor2hint** (*,*) *[real,private/allocatable]*
- rms/**corrmsl** *[mean_sd_type,private]*
- rms/**corrmslnr** *[mean_sd_2d_type,private]*
- rms/**difpol2hint** (*,*) *[real,allocatable/public]*
- rms/**difpollmr** (*,*) *[complex,allocatable/public]*
- rms/**difrmsl** *[mean_sd_type,private]*
- rms/**difrmslnr** *[mean_sd_2d_type,private]*
- rms/**diftor2hint** (*,*) *[real,allocatable/public]*
- rms/**dpdpc** (*,*) *[real,private/allocatable]*
- rms/**dpdtc** (*,*) *[real,private/allocatable]*
- rms/**dpkindrc** (*,*) *[real,private/allocatable]*
- rms/**dpkindrlm** (*) *[complex,private/allocatable]*
- rms/**dr_facc** (*) *[real,allocatable/public]*

- rms/**dtbpol2hint** (*,*) *[real,allocatable/public]*
- rms/**dtbpollmr** (*,*) *[complex,allocatable/public]*
- rms/**dtbrms_file** *[character,private]*
- rms/**dtbtor2hint** (*,*) *[real,allocatable/public]*
- rms/**dtvp** (*,*) *[real,private/allocatable]*
- rms/**dtvplm** (*) *[complex,private/allocatable]*
- rms/**dtvr** (*,*) *[real,private/allocatable]*
- rms/**dtvrlm** (*) *[complex,private/allocatable]*
- rms/**dtvrms_file** *[character,private]*
- rms/**dtvt** (*,*) *[real,private/allocatable]*
- rms/**dtvtlm** (*) *[complex,private/allocatable]*
- rms/**geo2hint** (*,*) *[real,private/allocatable]*
- rms/**geormsl** *[mean_sd_type,private]*
- rms/**geormslnr** *[mean_sd_2d_type,private]*
- rms/**iner2hint** (*,*) *[real,private/allocatable]*
- rms/**inerrmsl** *[mean_sd_type,private]*
- rms/**inerrmslnr** *[mean_sd_2d_type,private]*
- rms/**lf2hint** (*,*) *[real,private/allocatable]*
- rms/**lfp2** (*,*) *[real,private/allocatable]*
- rms/**lfp2lm** (*) *[complex,private/allocatable]*
- rms/**lfrlm** (*) *[complex,private/allocatable]*
- rms/**lfrmsl** *[mean_sd_type,private]*
- rms/**lfrmslnr** *[mean_sd_2d_type,private]*
- rms/**lft2** (*,*) *[real,private/allocatable]*
- rms/**lft2lm** (*) *[complex,private/allocatable]*
- rms/**mag2hint** (*,*) *[real,private/allocatable]*
- rms/**magrmsl** *[mean_sd_type,private]*
- rms/**magrmslnr** *[mean_sd_2d_type,private]*
- rms/**n_cheb_maxc** *[integer,public]*
  Number of Chebyshevs
- rms/**n_dtbrms_file** *[integer,private]*
- rms/**n_dtvrms_file** *[integer,private]*
- rms/**n_r_maxc** *[integer,public]*
  Number of radial points
- rms/**ncut** *[integer,public]*
  Number of points for the cut-off
- rms/**pfp2lm** (*) *[complex,private/allocatable]*

- rms/**pft2lm** (*) *[complex,private/allocatable]*
- rms/**plf2hint** (*,*) *[real,private/allocatable]*
- rms/**plfrmsl** *[mean_sd_type,private]*
- rms/**plfrmslnr** *[mean_sd_2d_type,private]*
- rms/**pre2hint** (*,*) *[real,private/allocatable]*
- rms/**prermsl** *[mean_sd_type,private]*
- rms/**prermslnr** *[mean_sd_2d_type,private]*
- rms/**rc** (*) *[real,private/allocatable]*
  Cut-off radii
- rms/**vp_old** (*,*,*) *[real,private/allocatable]*
- rms/**vr_old** (*,*,*) *[real,private/allocatable]*
- rms/**vt_old** (*,*,*) *[real,private/allocatable]*

Subroutines and functions

subroutine rms/**initialize_rms**()
Memory allocation of arrays used in the computation of r.m.s. force balance

Called from *magic*

Call to *init_rnb()*, *zerorms()*

subroutine rms/**finalize_rms**()
Deallocate arrays used for r.m.s. force balance computation

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 *initialize_rms()*, *output()*

subroutine rms/**init_rnb**(*r*, *rcut*, *rdea*, *r2*, *n_r_max2*, *n_cheb_max2*, *ns*, *rscheme_rms*)

Prepares the usage of a cut back radial grid where nS points on both boundaries are discarded. The aim actually is to discard boundary effects, but just not considering the boundary grid points does not work when you also want radial derivatives and integrals. For these we use the Chebychev transform which needs are particular number of grid points so that the fast cosine transform can be applied. Therefor more than just 2 points have to be thrown away, which may make sense anyway.

Parameters

- **r** (*n_r_max*) *[real,in]*
- **rcut** *[real,in]*
- **rdea** *[real,in]*
- **r2** (*) *[real,out,allocatable]*
- **n_r_max2** *[integer,out]* :: ‘)
- **n_cheb_max2** *[integer,out]*
- **ns** *[integer,out]* :: ‘)

- **rscheme_rms** *[real]*

Called from *initialize_rms()*

Call to *abortrun()*

subroutine rms/**get_nl_rms**(*nr*, *vr*, *vt*, *vp*, *dvrdr*, *dvrdt*, *dvrdp*, *dvtdr*, *dvtdp*, *dvpdr*, *dvpdp*, *cvr*, *advt*, *advp*, *lft*, *lfp*, *tscheme*, *lrmscalc*)

This subroutine computes the r.m.s. force balance terms which need to be computed on the grid

Parameters

- **nr** *[integer,in]* :: radial level
- **vr** (,) *[real,in]*
- **vt** (,) *[real,in]*
- **vp** (,) *[real,in]*
- **dvrdr** (,) *[real,in]*
- **dvrdt** (,) *[real,in]*
- **dvrdp** (,) *[real,in]*
- **dvtdr** (,) *[real,in]*
- **dvtdp** (,) *[real,in]*
- **dvpdr** (,) *[real,in]*
- **dvpdp** (,) *[real,in]*
- **cvr** (,) *[real,in]*
- **advt** (,) *[real,in]*
- **advp** (,) *[real,in]*
- **lft** (,) *[real,in]*
- **lfp** (,) *[real,in]*
- **tscheme** *[real]* :: time scheme
- **lrmscalc** *[logical,in]*

Called from *radialloop()*

Call to *get_openmp_blocks()*

subroutine rms/**transform_to_grid_rms**(*nr*, *p_rloc*)

This subroutine is used to transform the arrays used in r.m.s. force calculations from the spectral space to the physical grid.

Parameters

- **nr** *[integer,in]* :: radial level
- **p_rloc** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,inout]* :: pressure in LM space

Called from *transform_to_grid_space()*

Call to *scal_to_grad_spat()*

subroutine rms/**transform_to_lm_rms**(*nr*, *lfr*)

This subroutine is used to transform the arrays used in r.m.s. force calculations from the physical grid to spectral space.

Parameters

- **nr** *[integer,in]* :: radial level
- **lfr** (,) *[real,inout]* :: radial component of the Lorentz force

Called from *transform_to_lm_space()*

Call to *scal_to_sh()*, *spat_to_sphertor()*, *spat_to_qst()*

subroutine rms/**compute_lm_forces**(*nr*, *w_rloc*, *dw_rloc*, *ddw_rloc*, *z_rloc*, *s_rloc*, *xi_rloc*, *p_rloc*, *dp_rloc*, *advrlm*)

This subroutine finalizes the computation of the r.m.s. spectra once the quantities are back in spectral space.

Parameters

- **nr** *[integer,in]*
- **w_rloc** (*) *[complex,in]*
- **dw_rloc** (*) *[complex,in]* :: phi-deriv of dw/dr
- **ddw_rloc** (*) *[complex,in]*
- **z_rloc** (*) *[complex,in]*
- **s_rloc** (*) *[complex,in]*
- **xi_rloc** (*) *[complex,in]*
- **p_rloc** (*) *[complex,in]*
- **dp_rloc** (*) *[complex,in]*
- **advrlm** (*) *[complex,in]*

Called from *radialloop()*

Call to *hintrms()*

subroutine rms/**get_force**(*force2hint*, *forcerms*, *forcermsl*, *forcermslnr*, *volc*, *nrms_sets*, *timepassed*, *timenorm*, *l_stop_time*[, *forcepol2hint*[, *forcetor2hint*]])

This subroutine is used to compute the contributions of the forces in the Navier-Stokes equation

Parameters

- **force2hint** (1 + *l_max*,*n_r_max*) *[real,in]*
- **forcerms** *[real,out]*
- **forcermsl** *[mean_sd_type,inout]*
- **forcermslnr** *[mean_sd_2d_type,inout]*
- **volc** *[real,in]*
- **nrms_sets** *[integer,in]*

- **timepassed** *[real,in]*
- **timenorm** *[real,in]*
- **l_stop_time** *[logical,in]*
- **forcepol2hint** (1 + *l_max*,*n_r_max*) *[real,in,]*
- **forcetor2hint** (1 + *l_max*,*n_r_max*) *[real,in,]*

Called from *dtvrms()*

Call to *rint_r()*

subroutine rms/**dtvrms**(*time*, *nrms_sets*, *timepassed*, *timenorm*, *l_stop_time*)

This routine calculates and stores the different contributions of the forces entering the Navier-Stokes equation.

Parameters

- **time** *[real,in]*
- **nrms_sets** *[integer,inout]*
- **timepassed** *[real,in]*
- **timenorm** *[real,in]*
- **l_stop_time** *[logical,in]*

Called from *output()*

Call to *get_dr_rloc()*, *hint2dpol()*, *get_force()*

subroutine rms/**dtbrms**(*time*)

Parameters **time** *[real,in]*

Called from *output()*

Call to *get_poltorrms()*, *get_dr_rloc()*, *hint2dpollm()*, *rint_r()*

10.19.2 RMS_helpers.f90

Description

This module contains several useful subroutines required to compute RMS diagnostics

Quick access

Routines *get_poltorrms()*, *hint2dpol()*, *hint2dpollm()*, *hint2pol()*, *hint2pollm()*, *hint2tor()*, *hint2torlm()*, *hintrms()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod*: This module contains the blocking information
- *communications* (*reduce_radial()*): This module contains the different MPI communicators used in MagIC.
- *truncation* (*l_max()*, *lm_max_dtb()*, *n_r_max()*, *lm_max()*): This module defines the grid points and the truncation
- *radial_functions* (*or2()*, `rscheme_oc()`, *r()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *useful* (*cc2real()*): This module contains several useful routines.
- *integration* (*rint_r()*): Radial integration functions
- *lmmapping* (`mappings()`)
- *constants* (*vol_oc()*, *one()*): module containing constants and parameters used in the code.

Variables

Subroutines and functions

subroutine `rms_helpers/`**get_poltorrms**(*pol*, *drpol*, *tor*, *llm*, *ulm*, *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** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]* :: Poloidal field Potential
- **drpol** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]* :: Radial derivative of Pol
- **tor** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]* :: Toroidal field Potential
- **polrms** *[real,out]*
- **torrms** *[real,out]*
- **polasrms** *[real,out]*
- **torasrms** *[real,out]*
- **map** *[mappings,in]*

Options

- **llm** *[integer,in,optional/default=(-1 + shape(pol, 0) - ulm) / (-1)]*
- **ulm** *[integer,in,optional/default=-1 + llm + shape(pol, 0)]*

Called from *dtbrms()*

Call to *cc2real()*, *rint_r()*

subroutine `rms_helpers/`**hint2dpol**(*dpol*, *lmstart*, *lmstop*, *pol2hint*, *map*)

Parameters

- **dpol** (1 - lmstart + lmstop) *[complex,in]* :: Toroidal field Potential
- **pol2hint** (1 + *l_max*) *[real,inout]*
- **map** *[mappings,in]*

Options

- **lmstart** *[integer,in,optional/default=(-1 - lmstop + shape(dpol, 0)) / (-1)]*
- **lmstop** *[integer,in,optional/default=-1 + lmstart + shape(dpol, 0)]*

Called from *dtvrms()*

Call to *cc2real()*

subroutine rms_helpers/**hint2dpollm**(*dpol*, *lmstart*, *lmstop*, *pol2hint*, *map*)

Parameters

- **dpol** (1 - lmstart + lmstop) *[complex,in]*
- **pol2hint** (1 - lmstart + lmstop) *[real,inout]*
- **map** *[mappings,in]*

Options

- **lmstart** *[integer,in,optional/default=(-1 - lmstop + shape(dpol, 0)) / (-1)]*
- **lmstop** *[integer,in,optional/default=-1 + lmstart + shape(dpol, 0)]*

Called from *dtbrms()*

Call to *cc2real()*

subroutine rms_helpers/**hint2pol**(*pol*, *lb*, *ub*, *nr*, *lmstart*, *lmstop*, *pollmr*, *pol2hint*, *map*)

Parameters

- **pol** (1 - lb + ub) *[complex,in]* :: Poloidal field Potential
- **nr** *[integer,in]*
- **lmstart** *[integer,in]*
- **lmstop** *[integer,in]*
- **pollmr** (1 - lb + ub) *[complex,out]*
- **pol2hint** (1 + *l_max*) *[real,inout]*
- **map** *[mappings,in]*

Options

- **lb** *[integer,in,optional/default=(-1 + shape(pol, 0) - ub) / (-1)]*
- **ub** *[integer,in,optional/default=-1 + lb + shape(pol, 0)]*

Called from *get_pol_rhs_imp()*, *get_pol_rhs_imp_ghost()*, *assemble_pol()*, *assemble_single()*, *get_single_rhs_imp()*

Call to *cc2real()*

subroutine rms_helpers/**hint2pollm**(*pol*, *lb*, *ub*, *nr*, *lmstart*, *lmstop*, *pollmr*, *pol2hint*, *map*)

Parameters

- **pol** (1 - lb + ub) *[complex,in]* :: Poloidal field Potential
- **nr** *[integer,in]*
- **lmstart** *[integer,in]*
- **lmstop** *[integer,in]*
- **pollmr** (1 - lb + ub) *[complex,out]*
- **pol2hint** (1 - lb + ub) *[real,inout]*
- **map** *[mappings,in]*

Options

- **lb** *[integer,in,optional/default=(-1 + shape(pol, 0) - ub) / (-1)]*
- **ub** *[integer,in,optional/default=-1 + lb + shape(pol, 0)]*

Called from *get_mag_rhs_imp()*, *get_mag_rhs_imp_ghost()*

Call to *cc2real()*

subroutine rms_helpers/**hintrms**(*f*, *nr*, *lmstart*, *lmstop*, *lmp*, *f2hint*, *map*, *sphertor*)

Parameters

- **f** (*) *[complex,in]*
- **nr** *[integer,in]*
- **lmstart** *[integer,in]*
- **lmstop** *[integer,in]*
- **lmp** *[integer,in]*
- **f2hint** (1 + *l_max*) *[real,inout]*
- **map** *[mappings,in]*
- **sphertor** *[logical,in]*

Called from *compute_lm_forces()*

Call to *cc2real()*

subroutine rms_helpers/**hint2tor**(*tor*, *lb*, *ub*, *nr*, *lmstart*, *lmstop*, *tor2hint*, *map*)

Parameters

- **tor** (1 - lb + ub) *[complex,in]* :: Toroidal field Potential
- **nr** *[integer,in]*
- **lmstart** *[integer,in]*
- **lmstop** *[integer,in]*
- **tor2hint** (1 + *l_max*) *[real,inout]*
- **map** *[mappings,in]*

Options

- **lb** *[integer,in,optional/default=(-1 + shape(tor, 0) - ub) / (-1)]*

- **ub** *[integer,in,optional/default=-1 + lb + shape(tor, 0)]*

Called from *get_tor_rhs_imp()*, *get_tor_rhs_imp_ghost()*

Call to *cc2real()*

subroutine rms_helpers/**hint2torlm**(*tor*, *lb*, *ub*, *nr*, *lmstart*, *lmstop*, *tor2hint*, *map*)

Parameters

- **tor** (1 - lb + ub) *[complex,in]* :: Toroidal field Potential
- **nr** *[integer,in]*
- **lmstart** *[integer,in]*
- **lmstop** *[integer,in]*
- **tor2hint** (1 - lb + ub) *[real,inout]*
- **map** *[mappings,in]*

Options

- **lb** *[integer,in,optional/default=(-1 + shape(tor, 0) - ub) / (-1)]*
- **ub** *[integer,in,optional/default=-1 + lb + shape(tor, 0)]*

Called from *get_mag_rhs_imp()*, *get_mag_rhs_imp_ghost()*

Call to *cc2real()*

10.19.3 dtB.f90

Description

This module contains magnetic field stretching and advection terms plus a separate omega-effect. It is used for movie output.

Quick access

Variables *dtb_lmloc_container*, *dtb_rloc_container*, *padvlm*, *padvlm_lmloc*, *padvlm_rloc*, *padvlmic*, *padvlmic_lmloc*, *pdiflm*, *pdiflm_lmloc*, *pdiflmic*, *pdiflmic_lmloc*, *pstrlm*, *pstrlm_lmloc*, *pstrlm_rloc*, *tadvlm*, *tadvlm_lmloc*, *tadvlm_rloc*, *tadvlmic*, *tadvlmic_lmloc*, *tadvrlm_lmloc*, *tadvrlm_rloc*, *tdiflm*, *tdiflm_lmloc*, *tdiflmic*, *tdiflmic_lmloc*, *tomelm*, *tomelm_lmloc*, *tomelm_rloc*, *tomerlm_lmloc*, *tomerlm_rloc*, *tstrlm*, *tstrlm_lmloc*, *tstrlm_rloc*, *tstrrlm_lmloc*, *tstrrlm_rloc*

Routines *dtb_gather_lo_on_rank0()*, *finalize_dtb_mod()*, *get_dh_dtblm()*, *get_dtblm()*, *get_dtblmfinish()*, *initialize_dtb_mod()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod*: This module contains the blocking information
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *truncation* (*n_r_maxmag()*, *n_r_ic_maxmag()*, *n_r_max()*, *lm_max_dtb()*, *n_r_max_dtb()*, *n_r_ic_max_dtb()*, *lm_max()*, *n_cheb_max()*, *n_r_ic_max()*, *l_max()*, *n_phi_max()*, *ldtbmem()*, *l_axi()*, *n_theta_max()*, *nlat_padded()*): This module defines the grid points and the truncation
- *communications* (*gather_all_from_lo_to_rank0()*, *gt_oc()*, *gt_ic()*): This module contains the different MPI communicators used in MagIC.
- *mpi_transp_mod* (type_mpitransp()): This is an abstract class that will be used to define MPI transposers The actual implementation is deferred to either point-to-point (MPI_Isend and MPI_IRecv) communications or all-to-all (MPI_AlltoAll)
- mpi_ptop_mod (type_mpiptop()): This module contains the implementation of MPI_Isend/MPI_Irecv global transpose
- *physical_parameters* (*opm()*, *o_sr()*): Module containing the physical parameters
- *radial_functions* (*o_r_ic()*, *lambda()*, *or2()*, *dllambda()*, rscheme_oc(), *or1()*, *orho1()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *radial_data* (*nrstart()*, *nrstop()*): This module defines the MPI decomposition in the radial direction.
- *horizontal_data* (*dphi()*, *dlh()*, *hdif_b()*, *osn2()*, *cosn2()*, *osn1()*, *dtheta1s()*, *dtheta1a()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *logic* (*l_cond_ic()*, *l_dtrmagspec()*, *l_dtbmovie()*): Module containing the logicals that control the run
- *blocking* (*lo_map()*, *st_map()*, *l2lmas()*, *lm2l()*, *lm2m()*, *lmp2lmps()*, *lmp2lmpa()*, *lm2lmp()*, *llmmag()*, *ulmmag()*, *llm()*, *ulm()*): Module containing blocking information
- *radial_spectra*
- *sht* (*scal_to_sh()*)
- *constants* (*two()*): module containing constants and parameters used in the code.
- *radial_der* (*get_dr()*): Radial derivatives functions

Variables

- dtb_mod/**dtb_lmloc_container** (*,*,*) *[complex,private/target/allocatable]*
- dtb_mod/**dtb_rloc_container** (*,*,*) *[complex,private/target/allocatable]*
- dtb_mod/**padvlm** (*,*) *[complex,allocatable/public]*
- dtb_mod/**padvlm_lmloc** (*,*) *[complex,pointer/public]*
- dtb_mod/**padvlm_rloc** (*,*) *[complex,private/pointer]*
- dtb_mod/**padvlmic** (*,*) *[complex,allocatable/public]*
- dtb_mod/**padvlmic_lmloc** (*,*) *[complex,allocatable/public]*
- dtb_mod/**pdiflm** (*,*) *[complex,allocatable/public]*

- dtb_mod/**pdiflm_lmloc** (*,*) *[complex,allocatable/public]*
- dtb_mod/**pdiflmic** (*,*) *[complex,allocatable/public]*
- dtb_mod/**pdiflmic_lmloc** (*,*) *[complex,allocatable/public]*
- dtb_mod/**pstrlm** (*,*) *[complex,allocatable/public]*
- dtb_mod/**pstrlm_lmloc** (*,*) *[complex,pointer/public]*
- dtb_mod/**pstrlm_rloc** (*,*) *[complex,private/pointer]*
- dtb_mod/**tadvlm** (*,*) *[complex,allocatable/public]*
- dtb_mod/**tadvlm_lmloc** (*,*) *[complex,pointer/public]*
- dtb_mod/**tadvlm_rloc** (*,*) *[complex,private/pointer]*
- dtb_mod/**tadvlmic** (*,*) *[complex,allocatable/public]*
- dtb_mod/**tadvlmic_lmloc** (*,*) *[complex,allocatable/public]*
- dtb_mod/**tadvrlm_lmloc** (*,*) *[complex,pointer/public]*
- dtb_mod/**tadvrlm_rloc** (*,*) *[complex,private/pointer]*
- dtb_mod/**tdiflm** (*,*) *[complex,allocatable/public]*
- dtb_mod/**tdiflm_lmloc** (*,*) *[complex,allocatable/public]*
- dtb_mod/**tdiflmic** (*,*) *[complex,allocatable/public]*
- dtb_mod/**tdiflmic_lmloc** (*,*) *[complex,allocatable/public]*
- dtb_mod/**tomelm** (*,*) *[complex,allocatable/public]*
- dtb_mod/**tomelm_lmloc** (*,*) *[complex,pointer/public]*
- dtb_mod/**tomelm_rloc** (*,*) *[complex,private/pointer]*
- dtb_mod/**tomerlm_lmloc** (*,*) *[complex,pointer/public]*
- dtb_mod/**tomerlm_rloc** (*,*) *[complex,private/pointer]*
- dtb_mod/**tstrlm** (*,*) *[complex,allocatable/public]*
- dtb_mod/**tstrlm_lmloc** (*,*) *[complex,pointer/public]*
- dtb_mod/**tstrlm_rloc** (*,*) *[complex,private/pointer]*
- dtb_mod/**tstrrlm_lmloc** (*,*) *[complex,pointer/public]*
- dtb_mod/**tstrrlm_rloc** (*,*) *[complex,private/pointer]*

Subroutines and functions

subroutine dtb_mod/**initialize_dtb_mod**()
Memory allocation

Called from *magic*

subroutine dtb_mod/**finalize_dtb_mod**()
Memory deallocation

Called from *magic*

subroutine dtb_mod/**dtb_gather_lo_on_rank0**()

MPI gather on rank0 for dtBmovie outputs. This routine should really be suppressed once the movie outputs have been improved

Called from *get_dtblmfinish()*

Call to *gather_all_from_lo_to_rank0()*

subroutine dtb_mod/**get_dtblm**(*nr*, *vr*, *vt*, *vp*, *br*, *bt*, *bp*, *btvrlm*, *bpvrlm*, *brvtlm*, *brvplm*, *btvplm*, *bpvtlm*, *brvzlm*, *btvzlm*, *btvpcotlm*, *bpvtcotlm*, *btvzcotlm*, *btvpsn2lm*, *bpvtsn2lm*, *btvzsn2lm*)

This subroutine calculates non-linear products in grid-space for radial level nR.

Parameters

- **nr** *[integer,in]*
- **vr** (,) *[real,in]*
- **vt** (,) *[real,in]*
- **vp** (,) *[real,in]*
- **br** (,) *[real,in]*
- **bt** (,) *[real,in]*
- **bp** (,) *[real,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 *radialloop()*

Call to *scal_to_sh()*

subroutine dtb_mod/**get_dh_dtblm**(*nr*, *btvrlm*, *bpvrlm*, *brvtlm*, *brvplm*, *btvplm*, *bpvtlm*, *brvzlm*, *btvzlm*, *btvpcotlm*, *bpvtcotlm*, *btvpsn2lm*, *bpvtsn2lm*)

Purpose of this routine is to calculate theta and phi derivative related terms of the magnetic production and advection terms and store them.

Parameters

- **nr** *[integer,in]*
- **btvrlm** (*) *[complex,in]*
- **bpvrlm** (*) *[complex,in]*
- **brvtlm** (*) *[complex,in]*
- **brvplm** (*) *[complex,in]*
- **btvplm** (*) *[complex,in]*
- **bpvtlm** (*) *[complex,in]*
- **brvzlm** (*) *[complex,in]*
- **btvzlm** (*) *[complex,in]*
- **btvpcotlm** (*) *[complex,in]*
- **bpvtcotlm** (*) *[complex,in]*
- **btvpsn2lm** (*) *[complex,in]*
- **bpvtsn2lm** (*) *[complex,in]*

Called from *radialloop()*

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*, *l_frame*)

– Input of variables:

Parameters

- **time** *[real,in]*
- **n_time_step** *[integer,in]*
- **omega_ic** *[real,in]*
- **b** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]*
- **ddb** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]*
- **aj** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]*
- **dj** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]*
- **ddj** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]*
- **b_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,in]*
- **db_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,in]*
- **ddb_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,in]*
- **aj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,in]*
- **dj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,in]*
- **ddj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,in]*
- **l_frame** *[logical,in]*

Called from *output()*

Call to *rbrspec()*, *rbpspec()*, *dtb_gather_lo_on_rank0()*

10.19.4 dtB_arrays.f90

Needed modules

- *truncation* (*lmp_max_dtb()*): This module defines the grid points and the truncation
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *precision_mod*: This module controls the precision used in MagIC
- *constants* (*zero()*): module containing constants and parameters used in the code.

Types

- **type** dtb_arrays_mod/**unknown_type**

  Type fields

  - **% bpvrlm** (*) *[complex,allocatable]*
  - **% bpvtcotlm** (*) *[complex,allocatable]*
  - **% bpvtlm** (*) *[complex,allocatable]*
  - **% bpvtsn2lm** (*) *[complex,allocatable]*
  - **% brvplm** (*) *[complex,allocatable]*
  - **% brvtlm** (*) *[complex,allocatable]*
  - **% brvzlm** (*) *[complex,allocatable]*
  - **% btvpcotlm** (*) *[complex,allocatable]*
  - **% btvplm** (*) *[complex,allocatable]*
  - **% btvpsn2lm** (*) *[complex,allocatable]*
  - **% btvrlm** (*) *[complex,allocatable]*
  - **% btvzcotlm** (*) *[complex,allocatable]*
  - **% btvzlm** (*) *[complex,allocatable]*
  - **% btvzsn2lm** (*) *[complex,allocatable]*

Subroutines and functions

subroutine dtb_arrays_mod/**initialize**(*this*)

Parameters **this** *[real]*

subroutine dtb_arrays_mod/**finalize**(*this*)

Parameters **this** *[real]*

subroutine dtb_arrays_mod/**set_zero**(*this*)

Parameters **this** *[real]*

10.19.5 out_dtB_frame.f90

Quick access

Routines *get_bpol()*, *get_btor()*, *get_dtb()*, *lm2pt()*, *write_dtb_frame()*

Needed modules

- *truncation*: This module defines the grid points and the truncation
- *precision_mod*: This module controls the precision used in MagIC
- *radial_functions* (*r()*, *or1()*, *chebt_ic()*, *r_ic()*, rscheme_oc(), *r_icb()*, *dr_fac_ic()*, *chebt_ic_even()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *blocking* (*lm2m()*, *lm2l()*, *lm2()*): Module containing blocking information
- *horizontal_data* (*costheta()*, *n_theta_cal2ord()*, *sintheta()*, *osn1()*, *dlh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *dtb_mod* (*pstrlm()*, *padvlm()*, *pdiflm()*, *tstrlm()*, *tadvlm()*, *tdiflm()*, *padvlmic()*, *pdiflmic()*, *tadvlmic()*, *tomelm()*, *tdiflmic()*): This module contains magnetic field stretching and advection terms plus a separate omega-effect. It is used for movie output. . . .
- *movie_data* (*n_movie_type()*, *n_movie_fields()*, *n_movie_fields_ic()*, *n_movie_file()*, *n_movie_const()*, *n_movie_surface()*, *movie_const()*, *n_movie_field_type()*)
- *logic* (*l_cond_ic()*): Module containing the logicals that control the run
- *constants* (*zero()*, *one()*, *ci()*): module containing constants and parameters used in the code.
- *radial_der_even* (*get_drns_even()*)
- *radial_der* (*get_dr()*): Radial derivatives functions
- *sht* (*torpol_to_spat()*, *sphtor_to_spat()*, *scal_to_spat()*, *toraxi_to_spat()*)

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_bpol()*, *get_btor()*, *lm2pt()*, *get_drns_even()*

subroutine out_dtb_frame/**get_dtb**(*dtb*, *dtblm*, *dimb1*, *dimb2*, *n_r*, *l_ic*)

Parameters

- **dtb** (*) *[real,out]* :: Result Field with dim>=n_theta_block
- **dtblm** (dimb1,dimb2) *[complex,in]*
- **dimb1** *[integer,in,]*
- **dimb2** *[integer,in,]*
- **n_r** *[integer,in]* :: No. of radial grid point
- **l_ic** *[logical,in]* :: =true if inner core field

Called from *write_dtb_frame()*

Call to *toraxi_to_spat()*

subroutine out_dtb_frame/**get_bpol**(*pollm*, *dpollm*, *br*, *bt*, *bp*, *rt*, *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** (,) *[real,out]*
- **bt** (,) *[real,out]*
- **bp** (,) *[real,out]*
- **rt** *[real,in]* :: radius
- **lic** *[logical,in]* :: true for inner core, special rDep !

Called from *write_dtb_frame()*

Call to *torpol_to_spat()*

subroutine out_dtb_frame/**get_btor**(*tlm*, *bt*, *bp*, *rt*, *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** (,) *[real,out]*
- **bp** (,) *[real,out]*
- **rt** *[real,in]* :: radius
- **lic** *[logical,in]* :: true for inner core, special rDep !

Called from *write_dtb_frame()*

Call to *sphtor_to_spat()*

subroutine out_dtb_frame/**lm2pt**(*alm*, *aij*, *rt*, *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** (,) *[real,out]* :: field in (theta,phi)-space
- **rt** *[real,in]*
- **lic** *[logical,in]* :: true for inner core, extra factor !
- **lrcomp** *[logical,in]* :: true for radial field components

Called from *write_dtb_frame()*

Call to *scal_to_spat()*

10.19.6 TO.f90

Description

This module contains information for TO calculation and output

Quick access

Variables *bplast*, *bpsdas_rloc*, *bpzas_rloc*, *bpzdas_rloc*, *bs2as_rloc*, *bslast*, *bspas_rloc*, *bspdas_rloc*, *bszas_rloc*, *bzlast*, *bzpdas_rloc*, *ddzasl*, *dzasl*, *dzastras_rloc*, *dzcoras_rloc*, *dzddvpas_rloc*, *dzddvplmr*, *dzdvpas_rloc*, *dzdvplmr*, *dzlfas_rloc*, *dzrstras_rloc*, *dzstras_rloc*, *v2as_rloc*, *vas_rloc*, *zasl*

Routines *finalize_to()*, *get_pas()*, *getto()*, *gettofinish()*, *gettonext()*, *initialize_to()*, *prep_to_axi()*

Needed modules

- iso_fortran_env (output_unit())
- *precision_mod*: This module controls the precision used in MagIC
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *truncation* (*n_phi_maxstr()*, *n_r_maxstr()*, *l_max()*, *n_theta_maxstr()*, *lm_max()*, *nlat_padded()*): This module defines the grid points and the truncation
- *radial_data* (*n_r_cmb()*, *nrstart()*, *nrstop()*): This module defines the MPI decomposition in the radial direction.
- *radial_functions* (*r()*, *or1()*, *or2()*, *or3()*, *or4()*, *beta()*, *orho1()*, *dbeta()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *physical_parameters* (*corfac()*, *kbotv()*, *ktopv()*): Module containing the physical parameters
- *blocking* (*lm2()*, *llmmag()*, *ulmmag()*, *lo_map()*, *lm2l()*, *lm2m()*): Module containing blocking information
- *horizontal_data* (*sintheta()*, *costheta()*, *hdif_v()*, *dtheta1a()*, *dtheta1s()*, *dlh()*, *n_theta_cal2ord()*, *o_sin_theta()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *constants* (*one()*, *two()*): module containing constants and parameters used in the code.
- *logic* (*lverbose()*, *l_mag()*, *l_parallel_solve()*): Module containing the logicals that control the run
- *sht* (*spat_to_sh_axi()*, *toraxi_to_spat()*)

Variables

- torsional_oscillations/**bplast** (*,*,*) *[real,private/allocatable]*
- torsional_oscillations/**bpsdas_rloc** (*,*) *[real,allocatable/public]*
- torsional_oscillations/**bpzas_rloc** (*,*) *[real,allocatable/public]*
- torsional_oscillations/**bpzdas_rloc** (*,*) *[real,allocatable/public]*
- torsional_oscillations/**bs2as_rloc** (*,*) *[real,allocatable/public]*
- torsional_oscillations/**bslast** (*,*,*) *[real,private/allocatable]*
- torsional_oscillations/**bspas_rloc** (*,*) *[real,allocatable/public]*
- torsional_oscillations/**bspdas_rloc** (*,*) *[real,allocatable/public]*
- torsional_oscillations/**bszas_rloc** (*,*) *[real,allocatable/public]*
- torsional_oscillations/**bzlast** (*,*,*) *[real,private/allocatable]*
- torsional_oscillations/**bzpdas_rloc** (*,*) *[real,allocatable/public]*
- torsional_oscillations/**ddzasl** (*,*) *[real,allocatable/public]*
- torsional_oscillations/**dzasl** (*) *[real,private/allocatable]*
- torsional_oscillations/**dzastras_rloc** (*,*) *[real,allocatable/public]*
- torsional_oscillations/**dzcoras_rloc** (*,*) *[real,allocatable/public]*
- torsional_oscillations/**dzddvpas_rloc** (*,*) *[real,allocatable/public]*

- torsional_oscillations/**dzddvplmr** (*,*) *[real,private/allocatable]*
- torsional_oscillations/**dzdvpas_rloc** (*,*) *[real,allocatable/public]*
- torsional_oscillations/**dzdvplmr** (*,*) *[real,private/allocatable]*
- torsional_oscillations/**dzlfas_rloc** (*,*) *[real,allocatable/public]*
- torsional_oscillations/**dzrstras_rloc** (*,*) *[real,allocatable/public]*
- torsional_oscillations/**dzstras_rloc** (*,*) *[real,allocatable/public]*
- torsional_oscillations/**v2as_rloc** (*,*) *[real,allocatable/public]*
- torsional_oscillations/**vas_rloc** (*,*) *[real,allocatable/public]*
- torsional_oscillations/**zasl** (*) *[real,private/allocatable]*

Subroutines and functions

subroutine torsional_oscillations/**initialize_to**()
Allocate the memory needed

Called from *magic*

subroutine torsional_oscillations/**finalize_to**()
Deallocate the memory

Called from *magic*

subroutine torsional_oscillations/**prep_to_axi**(*z*, *dz*)

Parameters

- **z** (*) *[complex,in]*
- **dz** (*) *[complex,in]*

Called from *radialloop()*

subroutine torsional_oscillations/**getto**(*vr*, *vt*, *vp*, *cvr*, *dvpdr*, *br*, *bt*, *bp*, *cbr*, *cbt*, *dzrstrlm*, *dzastrlm*, *dzcorlm*, *dzlflm*, *dtlast*, *nr*)

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** (,) *[real,in]*
- **vt** (,) *[real,in]*
- **vp** (,) *[real,in]*

- **cvr** (,) *[real,in]*
- **dvpdr** (,) *[real,in]*
- **br** (,) *[real,in]*
- **bt** (,) *[real,in]*
- **bp** (,) *[real,in]*
- **cbr** (,) *[real,in]*
- **cbt** (,) *[real,in]*
- **dzrstrlm** (2 + *l_max*) *[real,out]*
- **dzastrlm** (2 + *l_max*) *[real,out]*
- **dzcorlm** (2 + *l_max*) *[real,out]*
- **dzlflm** (2 + *l_max*) *[real,out]*
- **dtlast** *[real,in]* :: last time step
- **nr** *[integer,in]* :: radial grid point

Called from *radialloop()*

Call to *spat_to_sh_axi()*

subroutine torsional_oscillations/**gettonext**(*br*, *bt*, *bp*, *ltonext*, *ltonext2*, *dt*, *dtlast*, *nr*)

Preparing TO calculation by storing flow and magnetic field contribution to build time derivative.

Parameters

- **br** (,) *[real,in]*
- **bt** (,) *[real,in]*
- **bp** (,) *[real,in]*
- **ltonext** *[logical,in]*
- **ltonext2** *[logical,in]*
- **dt** *[real,in]*
- **dtlast** *[real,in]*
- **nr** *[integer,in]*

Called from *radialloop()*

subroutine torsional_oscillations/**gettofinish**(*nr*, *dtlast*, *dzrstrlm*, *dzastrlm*, *dzcorlm*, *dzlflm*)

This program was previously part of getTO(...) It has now been separated to get it out of the theta-block loop.

Parameters

- **nr** *[integer,in]*
- **dtlast** *[real,in]*
- **dzrstrlm** (2 + *l_max*) *[real,in]*

- **dzastrlm** (2 + *l_max*) *[real,in]*
- **dzcorlm** (2 + *l_max*) *[real,in]*
- **dzlflm** (2 + *l_max*) *[real,in]*

Called from *radialloop()*

Call to *get_pas()*

subroutine torsional_oscillations/**get_pas**(*tlm*, *bp*, *nr*)

Purpose of this subroutine is to calculate the axisymmetric component Bp of an axisymmetric toroidal field Tlm given in spherical harmonic space (1:lmax+1). Unscrambling of theta is also ensured

Parameters

- **tlm** (*) *[real,in]* :: field in (l,m)-space for rT
- **bp** (*) *[real,out]*
- **nr** *[integer,in]*

Called from *gettofinish()*

Call to *toraxi_to_spat()*

10.19.7 TO_arrays.f90

Needed modules

- *truncation* (*l_max()*): This module defines the grid points and the truncation
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *precision_mod*: This module controls the precision used in MagIC

Types

- **type** to_arrays_mod/**unknown_type**

  Type fields

  - **% dzastrlm** (*) *[real,allocatable]*
  - **% dzcorlm** (*) *[real,allocatable]*
  - **% dzlflm** (*) *[real,allocatable]*
  - **% dzrstrlm** (*) *[real,allocatable]*

Subroutines and functions

subroutine to_arrays_mod/**initialize**(*this*)

Parameters **this** *[real]*

subroutine to_arrays_mod/**finalize**(*this*)

Parameters **this** *[real]*

subroutine to_arrays_mod/**set_zero**(*this*)

Parameters **this** *[real]*

10.19.8 out_TO.f90

Description

This module handles the writing of TO-related outputs: zonal force balance and z-integrated terms. This is a re-implementation of the spectral method used up to MagIC 5.10, which formerly relies on calculation of Plm on the cylindrical grid. This was quite costly and not portable on large truncations. As such, a local 4th order method is preferred here.

Quick access

Variables *bpsdas*, *bpzas*, *bpzdas*, *bs2as*, *bspas*, *bspdas*, *bszas*, *bzpdas*, *cyl*, *dzastras*, *dzcoras*, *dzddvpas*, *dzdvpas*, *dzlfas*, *dzrstras*, *dzstras*, *h*, *movfile*, *n_nhs_file*, *n_s_max*, *n_s_otc*, *n_shs_file*, *n_to_file*, *n_tomov_file*, *ntomovsets*, *oh*, *tofile*, *v2as*, *vas*, *volcyl_oc*

Routines *cylmean()*, *finalize_outto_mod()*, *gather_from_rloc_to_rank0()*, *get_dds()*, *get_ds()*, *initialize_outto_mod()*, *interp_theta()*, *outto()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod*: This module contains the blocking information
- *constants* (*one()*, *two()*, *pi()*, *vol_oc()*): module containing constants and parameters used in the code.
- *truncation* (*n_r_max()*, *n_theta_max()*, *n_phi_max()*, *minc()*): This module defines the grid points and the truncation
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *num_param* (*tscale()*): Module containing numerical and control parameters
- *output_data* (*sdens()*, *zdens()*, *tag()*, *runid()*, *log_file()*, *n_log_file()*): This module contains the parameters for output control
- *radial_data* (*radial_balance()*, *nrstart()*, *nrstop()*): This module defines the MPI decomposition in the radial direction.

- *radial_functions* (*r_icb()*, *r_cmb()*, *r()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *horizontal_data* (*theta_ord()*, *phi()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *logic* (*l_save_out()*, *l_tomovie()*, *l_full_sphere()*): Module containing the logicals that control the run
- *physical_parameters* (*ra()*, *ek()*, *pr()*, *prmag()*, *radratio()*, *lffac()*): Module containing the physical parameters
- *torsional_oscillations* (*dzcoras_rloc()*, *dzdvpas_rloc()*, *dzddvpas_rloc()*, *dzrstras_rloc()*, *dzastras_rloc()*, *dzstras_rloc()*, *dzlfas_rloc()*, *v2as_rloc()*, *bs2as_rloc()*, *bspdas_rloc()*, *bpsdas_rloc()*, *bzpdas_rloc()*, *bpzdas_rloc()*, *bpzas_rloc()*, *bspas_rloc()*, *bszas_rloc()*, *vas_rloc()*): This module contains information for TO calculation and output
- *useful* (*logwrite()*): This module contains several useful routines.
- *integration* (*cylmean_otc()*, *cylmean_itc()*, *simps()*): Radial integration functions

Variables

- outto_mod/**bpsdas** (*,*) *[real,private/allocatable]*
- outto_mod/**bpzas** (*,*) *[real,private/allocatable]*
- outto_mod/**bpzdas** (*,*) *[real,private/allocatable]*
- outto_mod/**bs2as** (*,*) *[real,private/allocatable]*
- outto_mod/**bspas** (*,*) *[real,private/allocatable]*
- outto_mod/**bspdas** (*,*) *[real,private/allocatable]*
- outto_mod/**bszas** (*,*) *[real,private/allocatable]*
- outto_mod/**bzpdas** (*,*) *[real,private/allocatable]*
- outto_mod/**cyl** (*) *[real,private/allocatable]*
  Cylindrical grid
- outto_mod/**dzastras** (*,*) *[real,private/allocatable]*
- outto_mod/**dzcoras** (*,*) *[real,private/allocatable]*
- outto_mod/**dzddvpas** (*,*) *[real,private/allocatable]*
- outto_mod/**dzdvpas** (*,*) *[real,private/allocatable]*
- outto_mod/**dzlfas** (*,*) *[real,private/allocatable]*
- outto_mod/**dzrstras** (*,*) *[real,private/allocatable]*
- outto_mod/**dzstras** (*,*) *[real,private/allocatable]*
- outto_mod/**h** (*) *[real,private/allocatable]*
  height
- outto_mod/**movfile** *[character,private]*
- outto_mod/**n_nhs_file** *[integer,private]*
- outto_mod/**n_s_max** *[integer,private]*
- outto_mod/**n_s_otc** *[integer,private]*

- outto_mod/**n_shs_file** *[integer,private]*
- outto_mod/**n_to_file** *[integer,private]*
- outto_mod/**n_tomov_file** *[integer,private]*
- outto_mod/**ntomovsets** *[integer,private]*
  Number of TO_mov frames
- outto_mod/**oh** (*) *[real,private/allocatable]*
  1/h
- outto_mod/**tofile** *[character,private]*
- outto_mod/**v2as** (*,*) *[real,private/allocatable]*
- outto_mod/**vas** (*,*) *[real,private/allocatable]*
- outto_mod/**volcyl_oc** *[real,private]*

Subroutines and functions

subroutine outto_mod/**initialize_outto_mod**()
Memory allocation of arrays needed for TO outputs

Called from *magic*

Call to *simps()*

subroutine outto_mod/**finalize_outto_mod**()
Memory de-allocation of arrays needed for TO outputs

Called from *magic*

subroutine outto_mod/**outto**(*time*, *n_time_step*, *ekin*, *ekintas*, *ltomov*)

Output of axisymmetric zonal flow, its relative strength, its time variation, and all forces acting on it.

Parameters

- **time** *[real,in]* :: time
- **n_time_step** *[integer,in]* :: Iteration number
- **ekin** *[real,in]* :: Kinetic energy
- **ekintas** *[real,in]* :: Toroidal axisymmetric energy
- **ltomov** *[logical,in]* :: Do we need to store the movie files as well

Called from *output()*

Call to *gather_from_rloc_to_rank0()*, *cylmean()*, *interp_theta()*, *get_ds()*, *get_dds()*, *simps()*, *logwrite()*

subroutine outto_mod/**cylmean**(*dat*, *datn*, *dats*)

This routine computes the z-average inside and outside TC

Parameters

- **dat** (,) *[real,in]* :: input data
- **datn** (*n_s_max*) *[real,out]* :: z-average outside T.C. + N.H.

- **dats** (*n_s_max*) *[real,out]* :: z-average outside T.C. + S.H.

Called from *outgeos()*, *outto()*

Call to *cylmean_otc()*, *cylmean_itc()*

subroutine outto_mod/**interp_theta**(*a*, *ac*, *rr*, *cyl*, *theta*)

This routine computes the interpolation of a value at the surface of a spherical shell onto the cylindrical grid. This is only a theta interpolation using the cylindrical theta's.

Parameters

- **a** (*) *[real,in]* :: Field at the outer radius
- **ac** (2) *[real,out]* :: Surface values for NH and SH as a function of s
- **rr** *[real,in]* :: Radius at which we compute the extrapolation (can be ri or ro)
- **cyl** *[real,in]* :: Cylindrical radius
- **theta** (*) *[real,in]* :: Colatitude

Called from *outto()*

subroutine outto_mod/**get_ds**(*arr*, *darr*, *cyl*)

This subroutine is used to compute the 4th order accurate first s-derivative on the regularly spaced grid

Parameters

- **arr** (*) *[real,in]* :: Array to be differentiated
- **darr** (*) *[real,out]* :: s-derivative of the input array
- **cyl** (*) *[real,in]* :: Cylindrical grid

Called from *outto()*

subroutine outto_mod/**get_dds**(*arr*, *ddarr*, *cyl*)

This subroutine is used to compute the 4th order accurate 2nd s-derivative on the regularly spaced grid

https://bellaard.com/tools/Finite%20difference%20coefficient%20calculator/

Parameters

- **arr** (*) *[real,in]* :: Array to be differentiated
- **ddarr** (*) *[real,out]* :: s-derivative of the input array
- **cyl** (*) *[real,in]* :: Cylindrical grid

Called from *outto()*

subroutine outto_mod/**gather_from_rloc_to_rank0**(*arr_rloc*, *arr*)

This subroutine gathers the r-distributed array

Parameters

- **arr_rloc** (*n_theta_max*,1 - *nrstart* + *nrstop*) *[real,in]*
- **arr** (*n_theta_max*,*n_r_max*) *[real,out]*

Called from *outto()*

10.19.9 radial_spectra.f90

Quick access

Variables *filehandle*

Routines *rbpspec()*, *rbrspec()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod*: This module contains the blocking information
- *communications* (*reduce_radial()*): This module contains the different MPI communicators used in MagIC.
- *truncation* (*lm_max()*, *n_r_max()*, *n_r_ic_max()*, *l_max()*, *n_r_tot()*): This module defines the grid points and the truncation
- *radial_data* (*n_r_icb()*): This module defines the MPI decomposition in the radial direction.
- *radial_functions* (*or2()*, *r_icb()*, *r_ic()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *num_param* (*escale()*): Module containing numerical and control parameters
- *blocking* (*st_map()*, *llm()*, *ulm()*): Module containing blocking information
- *horizontal_data* (*dlh()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *logic* (*l_cond_ic()*): Module containing the logicals that control the run
- *output_data* (*tag()*): This module contains the parameters for output control
- *useful* (*cc2real()*): This module contains several useful routines.
- *lmmapping* (mappings())
- *constants* (*pi()*, *one()*, *four()*, *half()*): module containing constants and parameters used in the code.

Variables

- radial_spectra/**filehandle** *[integer,private]*

Subroutines and functions

subroutine radial_spectra/**rbrspec**(*time*, *pol*, *polic*, *fileroot*, *lic*, *map*)

Parameters

- **time** *[real,in]*
- **pol** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **polic** (1 - *llm* + *ulm*,*n_r_ic_max*) *[complex,in]*
- **fileroot** *[character,in]*
- **lic** *[logical,in]*
- **map** *[mappings,in]*

Called from *get_dtblmfinish()*, *output()*

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** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **toric** (1 - *llm* + *ulm*,*n_r_ic_max*) *[complex,in]*
- **fileroot** *[character,in]*
- **lic** *[logical,in]*
- **map** *[mappings,in]*

Called from *get_dtblmfinish()*, *output()*

Call to *cc2real()*

10.19.10 outGeos.f90

Description

This module is used to compute z-integrated diagnostics such as the degree of geostrophy or the separation of energies between inside and outside the tangent cylinder. This makes use of a local Simpson's method. This also handles the computation of the z-average profile of rotation when a Couette flow setup is used

Quick access

Variables *geos_file*, *n_geos_file*, *npstart*, *npstop*, *phi_balance*, *up_ploc*, *up_rloc*, *us_ploc*, *us_rloc*, *uz_ploc*, *uz_rloc*, *vol_otc*, *wz_ploc*, *wz_rloc*

Routines *calcgeos()*, *finalize_geos()*, *initialize_geos()*, *outgeos()*, *outomega()*, *transp_r2phi()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod*: This module contains the blocking information
- *blocking* (*lo_map()*, *llm()*, *ulm()*): Module containing blocking information
- *constants* (*half()*, *two()*, *pi()*, *one()*, *four()*, *third()*, *zero()*): module containing constants and parameters used in the code.
- *mem_alloc* (*bytes_allocated()*): This little module is used to estimate the global memory allocation used in MagIC
- *radial_data* (*radial_balance()*, *nrstart()*, *nrstop()*): This module defines the MPI decomposition in the radial direction.
- *radial_functions* (*or1()*, *or2()*, *r_icb()*, *r_cmb()*, *r()*, *orho1()*, *orho2()*, *beta()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *output_data* (*sdens()*, *zdens()*, *tag()*): This module contains the parameters for output control
- *horizontal_data* (*n_theta_cal2ord()*, *o_sin_theta_e2()*, *theta_ord()*, *o_sin_theta()*, *costheta()*, *sintheta()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *truncation* (*n_phi_max()*, *n_theta_max()*, *n_r_max()*, *nlat_padded()*, *l_max()*): This module defines the grid points and the truncation
- *integration* (*simps()*, *cylmean_otc()*, *cylmean_itc()*): Radial integration functions
- *logic* (*l_save_out()*): Module containing the logicals that control the run
- *sht* (*toraxi_to_spat()*)

Variables

- geos/**cyl** (*) *[real,private/allocatable]*
  Cylindrical grid
- geos/**geos_file** *[character,private]*
  file name
- geos/**h** (*) *[real,private/allocatable]*
  h(s)
- geos/**n_geos_file** *[integer,private]*
  file unit for geos.TAG
- geos/**n_s_max** *[integer,private]*
  Number of cylindrical points

- geos/**n_s_otc** *[integer,private]*
  Index for last point outside TC
- geos/**npstart** *[integer,private]*
  Starting nPhi index when MPI distributed
- geos/**npstop** *[integer,private]*
  Stoping nPhi index when MPI distributed
- geos/**phi_balance** (*) *[load,private/allocatable]*
  phi-distributed balance
- geos/**up_ploc** (*,*,*) *[real,private/allocatable]*
- geos/**up_rloc** (*,*,*) *[real,private/allocatable]*
- geos/**us_ploc** (*,*,*) *[real,private/allocatable]*
- geos/**us_rloc** (*,*,*) *[real,private/allocatable]*
- geos/**uz_ploc** (*,*,*) *[real,private/allocatable]*
- geos/**uz_rloc** (*,*,*) *[real,private/allocatable]*
- geos/**vol_otc** *[real,private]*
  volume outside tangent cylinder
- geos/**wz_ploc** (*,*,*) *[real,private/allocatable]*
- geos/**wz_rloc** (*,*,*) *[real,private/allocatable]*

Subroutines and functions

subroutine geos/**initialize_geos**(*l_geos*, *l_sric*)

Memory allocation and definition of the cylindrical grid

Parameters

- **l_geos** *[logical,in]* :: Do we need the geos outputs
- **l_sric** *[logical,in]* :: Is the inner core rotating

Called from *magic*

Call to *getblocks()*, *simps()*

subroutine geos/**finalize_geos**(*l_geos*, *l_sric*)

Memory deallocation

Parameters

- **l_geos** *[logical,in]* :: Do we need the geos outputs
- **l_sric** *[logical,in]* :: Is the inner core rotating?

Called from *magic*

subroutine geos/**calcgeos**(*vr*, *vt*, *vp*, *cvr*, *dvrdp*, *dvpdr*, *nr*)

This routine computes the term needed for geos.TAG outputs in physical space.

Parameters

- **vr** (,) *[real,in]*
- **vt** (,) *[real,in]*
- **vp** (,) *[real,in]*
- **cvr** (,) *[real,in]*
- **dvrdp** (,) *[real,in]*
- **dvpdr** (,) *[real,in]*
- **nr** *[integer,in]* :: Radial grid point

Called from *radialloop()*

subroutine geos/**outgeos**(*time*, *geos*, *geosa*, *geosz*, *geosm*, *geosnap*, *ekin*)

This routine handles the output of geos.TAG

Parameters

- **time** *[real,in]*
- **geos** *[real,out]*
- **geosa** *[real,out]*
- **geosz** *[real,out]*
- **geosm** *[real,out]*
- **geosnap** *[real,out]*
- **ekin** *[real,out]*

Called from *output()*

Call to *transp_r2phi()*, *cylmean()*, *simps()*, *cylmean_otc()*

subroutine geos/**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** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]* :: Toroidal potential
- **omega_ic** *[real,in]* :: Rotation rate of the inner core

Called from *output()*

Call to *toraxi_to_spat()*, *cylmean_otc()*, *cylmean_itc()*

subroutine geos/**transp_r2phi**(*arr_rloc*, *arr_ploc*)

This subroutine is used to compute a MPI transpose between a R-distributed array and a Phi-distributed array

Parameters

- **arr_rloc** (*n_theta_max*,*n_phi_max*,1 - *nrstart* + *nrstop*) *[real,in]*

- **arr_ploc** (*n_theta_max*,*n_r_max*,1 - *npstart* + *npstop*) *[real,out]*

Called from *outgeos()*

subroutine geos/**cylmean**(*dat*, *dat_otc*, *dat_itc_n*, *dat_itc_s*)

This routine computes the z-average inside and outside TC

Parameters

- **dat** (,) *[real,in]*
- **dat_otc** (*n_s_max*) *[real,out]*
- **dat_itc_n** (*n_s_max*) *[real,out]*
- **dat_itc_s** (*n_s_max*) *[real,out]*

Call to *cylmean_otc()*, *cylmean_itc()*

10.19.11 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_ekin_solid_liquid()*, *get_fluxes()*, *get_helicity()*, *get_nlblayers()*, *get_perppar()*, *get_visc_heat()*

Needed modules

- *precision_mod*: This module controls the precision used in MagIC
- *truncation* (*n_phi_max()*, *l_max()*, *l_maxmag()*, *n_theta_max()*): This module defines the grid points and the truncation
- *constants* (*pi()*, *one()*, *two()*, *third()*, *half()*): module containing constants and parameters used in the code.
- *logic* (*l_mag_nl()*, *l_anelastic_liquid()*): Module containing the logicals that control the run
- *physical_parameters* (*ek()*, *vischeatfac()*, *thexpnb()*): Module containing the physical parameters
- *radial_data* (*n_r_icb()*, *n_r_cmb()*): This module defines the MPI decomposition in the radial direction.
- *radial_functions* (*orho1()*, *orho2()*, *or2()*, *or1()*, *beta()*, *temp0()*, *visc()*, *or4()*, *r()*, *alpha0()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *horizontal_data* (*o_sin_theta_e2()*, *costheta()*, *sn2()*, *osn2()*, *cosn2()*, *gauss()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order

Variables

Subroutines and functions

subroutine nl_special_calc/**get_nlblayers**(*vt*, *vp*, *dvtdr*, *dvpdr*, *dsdr*, *dsdt*, *dsdp*, *uhas*, *duhas*, *gradsas*, *nr*)

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** (,) *[real,in]*
- **vp** (,) *[real,in]*
- **dvtdr** (,) *[real,in]*
- **dvpdr** (,) *[real,in]*
- **dsdr** (,) *[real,in]*
- **dsdt** (,) *[real,in]*
- **dsdp** (,) *[real,in]*
- **uhas** *[real,out]*
- **duhas** *[real,out]*
- **gradsas** *[real,out]*
- **nr** *[integer,in]*

Called from *radialloop()*

subroutine nl_special_calc/**get_perppar**(*vr*, *vt*, *vp*, *eperpas*, *eparas*, *eperpaxias*, *eparaxias*, *nr*)

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** (,) *[real,in]*
- **vt** (,) *[real,in]*
- **vp** (,) *[real,in]*
- **eperpas** *[real,out]*
- **eparas** *[real,out]*
- **eperpaxias** *[real,out]*
- **eparaxias** *[real,out]*

- **nr** *[integer,in]*

Called from *radialloop()*

subroutine nl_special_calc/**get_fluxes**(*vr*, *vt*, *vp*, *dvrdr*, *dvtdr*, *dvpdr*, *dvrdt*, *dvrdp*, *sr*, *pr*, *br*, *bt*, *bp*, *cbt*, *cbp*, *fconvas*, *fkinas*, *fviscas*, *fpoynas*, *fresas*, *nr*)

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** (,) *[real,in]*
- **vt** (,) *[real,in]*
- **vp** (,) *[real,in]*
- **dvrdr** (,) *[real,in]*
- **dvtdr** (,) *[real,in]*
- **dvpdr** (,) *[real,in]*
- **dvrdt** (,) *[real,in]*
- **dvrdp** (,) *[real,in]*
- **sr** (,) *[real,in]*
- **pr** (,) *[real,in]*
- **br** (,) *[real,in]*
- **bt** (,) *[real,in]*
- **bp** (,) *[real,in]*
- **cbt** (,) *[real,in]*
- **cbp** (,) *[real,in]*
- **fconvas** *[real,out]*
- **fkinas** *[real,out]*
- **fviscas** *[real,out]*
- **fpoynas** *[real,out]*
- **fresas** *[real,out]*
- **nr** *[integer,in]*

Called from *radialloop()*

subroutine nl_special_calc/**get_helicity**(*vr*, *vt*, *vp*, *cvr*, *dvrdt*, *dvrdp*, *dvtdr*, *dvpdr*, *helas*, *hel2as*, *helnaas*, *helna2as*, *heleaas*, *nr*)

Calculates axisymmetric contributions of helicity HelLMr and helicity**2 Hel2LMr in (l,m=0,r) space.

Parameters

- **vr** (,) *[real,in]*
- **vt** (,) *[real,in]*
- **vp** (,) *[real,in]*
- **cvr** (,) *[real,in]*
- **dvrdt** (,) *[real,in]*
- **dvrdp** (,) *[real,in]*
- **dvtdr** (,) *[real,in]*
- **dvpdr** (,) *[real,in]*
- **helas** (2) *[real,out]*
- **hel2as** (2) *[real,out]*
- **helnaas** (2) *[real,out]*
- **helna2as** (2) *[real,out]*
- **heleaas** *[real,out]*
- **nr** *[integer,in]*

Called from *radialloop()*

subroutine nl_special_calc/**get_visc_heat**(*vr*, *vt*, *vp*, *cvr*, *dvrdr*, *dvrdt*, *dvrdp*, *dvtdr*, *dvtdp*, *dvpdr*, *dvpdp*, *viscas*, *nr*)

Calculates axisymmetric contributions of the viscous heating

Parameters

- **vr** (,) *[real,in]*
- **vt** (,) *[real,in]* ::

(4)

- **vp** (,) *[real,in]*
- **cvr** (,) *[real,in]* ::

(6)

- **dvrdr** (,) *[real,in]* ::

(2)

- **dvrdt** (,) *[real,in]*
- **dvrdp** (,) *[real,in]*
- **dvtdr** (,) *[real,in]*

- **dvtdp** (,) *[real,in]*
- **dvpdr** (,) *[real,in]*
- **dvpdp** (,) *[real,in]*
- **viscas** *[real,out]*
- **nr** *[integer,in]*

Called from *radialloop()*

subroutine nl_special_calc/**get_ekin_solid_liquid**(*vr*, *vt*, *vp*, *phi*, *ekins*, *ekinl*, *vols*, *nr*)

This subroutine computes the kinetic energy content in the solid and in the liquid phase when phase field is employed.

Parameters

- **vr** (,) *[real,in]*
- **vt** (,) *[real,in]*
- **vp** (,) *[real,in]*
- **phi** (,) *[real,in]*
- **ekins** *[real,out]* :: Kinetic energy in the solid phase
- **ekinl** *[real,out]* :: Kinetic energy in the liquid phase
- **vols** *[real,out]* :: volume of the solid
- **nr** *[integer,in]*

Called from *radialloop()*

10.19.12 probes.f90

Quick access

Variables *n_phi_probes*, *n_probebr*, *n_probebt*, *n_probevp*, *n_theta_usr*, *probe_filebr*, *probe_filebt*, *probe_filevp*, *r_probe*, *rad_rank*, *rad_usr*, *theta_probe*

Routines *finalize_probes()*, *initialize_probes()*, *probe_out()*

Needed modules

- *parallel_mod* (rank()): This module contains the blocking information
- *precision_mod*: This module controls the precision used in MagIC
- *truncation* (*n_r_max()*, *n_phi_max()*, *n_theta_max()*): This module defines the grid points and the truncation
- *radial_data* (*nrstart()*, *nrstop()*): This module defines the MPI decomposition in the radial direction.
- *radial_functions* (*r_cmb()*, *orho1()*, *or1()*, *or2()*, *r()*, *r_icb()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *num_param* (*vscale()*): Module containing numerical and control parameters

- *horizontal_data* (*o_sin_theta()*, *theta()*): Module containing functions depending on longitude and latitude plus help arrays depending on degree and order
- *output_data* (*tag()*): This module contains the parameters for output control
- *constants* (*pi()*, *one()*): module containing constants and parameters used in the code.
- *logic* (*l_save_out()*): Module containing the logicals that control the run
- *physical_parameters* (*radratio()*): Module containing the physical parameters

Variables

- probe_mod/**n_phi_probes** *[integer,public]*
  number of probes in phi - symmetrically distributed
- probe_mod/**n_probebr** *[integer,private]*
- probe_mod/**n_probebt** *[integer,private]*
- probe_mod/**n_probevp** *[integer,private]*
- probe_mod/**n_theta_usr** *[integer,private]*
- probe_mod/**probe_filebr** *[character,private]*
- probe_mod/**probe_filebt** *[character,private]*
- probe_mod/**probe_filevp** *[character,private]*
- probe_mod/**r_probe** *[real,public]*
- probe_mod/**rad_rank** *[integer,private]*
- probe_mod/**rad_usr** *[integer,private]*
- probe_mod/**theta_probe** *[real,public]*
  probe locations, r_probe in terms of r_cmb and theta in degrees

Subroutines and functions

subroutine probe_mod/**initialize_probes**()

Called from *magic*

subroutine probe_mod/**finalize_probes**()

Called from *magic*

subroutine probe_mod/**probe_out**(*time*, *n_r*, *vp*, *br*, *bt*)

Parameters

- **time** *[real,in]*
- **n_r** *[integer,in]* :: radial grod point no.
- **vp** (,) *[real,in]*
- **br** (,) *[real,in]*
- **bt** (,) *[real,in]*

Called from *radialloop()*

10.20 Reading and storing check points (restart files)

10.20.1 readCheckPoints.f90

Description

This module contains the functions that can help reading and mapping of the restart files

Quick access

Variables *bytes_allocated*, *l_axi_old*, *lreadr*, *lreads*, *lreadxi*, *n_start_file*, *ratio1*, *ratio1_old*, *ratio2*, *ratio2_old*

Routines *finish_start_fields()*, *getlm2lmo()*, *mapdatahydro()*, *mapdatamag()*, *mapdatar()*, *maponefield()*, *maponefield_mpi()*, *read_map_one_field()*, *read_map_one_field_mpi()*, *read_map_one_scalar()*, *read_map_one_scalar_mpi()*, *readstartfields()*, *readstartfields_mpi()*, *readstartfields_old()*

Needed modules

- iso_fortran_env (output_unit())
- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod*: This module contains the blocking information
- *communications* (*scatter_from_rank0_to_lo()*, lo2r_one()): This module contains the different MPI communicators used in MagIC.
- *fields* (*dw_lmloc()*, *ddw_lmloc()*, *ds_lmloc()*, *dp_lmloc()*, *dz_lmloc()*, *dxi_lmloc()*, *db_lmloc()*, *ddb_lmloc()*, *dj_lmloc()*, *ddj_lmloc()*, *db_ic_lmloc()*, *ddb_ic_lmloc()*, *dj_ic_lmloc()*, *ddj_ic_lmloc()*): This module contains all the fields used in MagIC in the hybrid (LM,r) space as well as their radial derivatives. It defines both the LM-distributed arrays and the R-distributed arrays....
- *truncation* (*n_r_max()*, *lm_max()*, *n_r_maxmag()*, *lm_maxmag()*, *n_r_ic_max()*, *n_r_ic_maxmag()*, *nalias()*, *n_phi_tot()*, *l_max()*, *m_max()*, *minc()*, *lmagmem()*, *fd_stretch()*, *fd_ratio()*): This module defines the grid points and the truncation
- *logic* (*l_rot_ma()*, *l_rot_ic()*, *l_sric()*, *l_srma()*, *l_cond_ic()*, *l_heat()*, *l_mag()*, *l_mag_lf()*, *l_chemical_conv()*, *l_ab1()*, *l_bridge_step()*, *l_double_curl()*, *l_z10mat()*, *l_single_matrix()*, *l_parallel_solve()*, *l_mag_par_solve()*, *l_phase_field()*): Module containing the logicals that control the run
- *blocking* (*lo_map()*, *lm2l()*, *lm2m()*, *lm_balance()*, *llm()*, *ulm()*, *llmmag()*, *ulmmag()*, *st_map()*): Module containing blocking information
- *init_fields* (*start_file()*, *inform()*, *tomega_ic1()*, *tomega_ic2()*, *tomega_ma1()*, *tomega_ma2()*, *omega_ic1()*, *omegaosz_ic1()*, *omega_ic2()*, *omegaosz_ic2()*, *omega_ma1()*, *omegaosz_ma1()*, *omega_ma2()*, *omegaosz_ma2()*, *tshift_ic1()*, *tshift_ic2()*, *tshift_ma1()*, *tshift_ma2()*, *tipdipole()*, *scale_b()*, *scale_v()*, *scale_s()*, *scale_xi()*): This module is used to construct the initial solution.

- *radial_functions* (rscheme_oc(), *chebt_ic()*, *cheb_norm_ic()*, *r()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *num_param* (*alph1()*, *alph2()*, *alpha()*): Module containing numerical and control parameters
- *radial_data* (*n_r_icb()*, *n_r_cmb()*): This module defines the MPI decomposition in the radial direction.
- *physical_parameters* (*ra()*, *ek()*, *pr()*, *prmag()*, *radratio()*, *sigma_ratio()*, *kbotv()*, *ktopv()*, *sc()*, *raxi()*, *lffac()*): Module containing the physical parameters
- *constants* (*c_z10_omega_ic()*, *c_z10_omega_ma()*, *pi()*, *zero()*, *two()*, *one()*, *c_lorentz_ma()*, *c_lorentz_ic()*, *c_moi_ic()*, *c_moi_ma()*): module containing constants and parameters used in the code.
- *chebyshev* (type_cheb_odd())
- *radial_scheme* (type_rscheme()): This is an abstract type that defines the radial scheme used in MagIC
- *finite_differences* (type_fd()): This module is used to calculate the radial grid when finite differences are requested
- *cosine_transform_odd* (costf_odd_t()): This module contains the built-in type I discrete Cosine Transforms. This implementation is based on Numerical Recipes and FFTPACK. This only works for n_r_max-1 = 2**a 3**b 5**c, with a,b,c integers....
- *useful* (*polynomial_interpolation()*, *abortrun()*): This module contains several useful routines.
- *updatewp_mod* (*get_pol_rhs_imp()*): This module handles the time advance of the poloidal potential w and the pressure p. It contains the computation of the implicit terms and the linear solves.
- *updatez_mod* (*get_tor_rhs_imp()*): This module handles the time advance of the toroidal potential z It contains the computation of the implicit terms and the linear solves....
- *updates_mod* (*get_entropy_rhs_imp()*): This module handles the time advance of the entropy s. It contains the computation of the implicit terms and the linear solves....
- *updatexi_mod* (*get_comp_rhs_imp()*): This module handles the time advance of the chemical composition xi. It contains the computation of the implicit terms and the linear solves....
- *updateb_mod* (*get_mag_rhs_imp()*, *get_mag_ic_rhs_imp()*): This module handles the time advance of the magnetic field potentials b and aj as well as the inner core counterparts b_ic and aj_ic. It contains the computation of the implicit terms and the linear
- *updatewps_mod* (*get_single_rhs_imp()*): This module handles the time advance of the poloidal potential w, the pressure p and the entropy s in one single matrix per degree. It contains the computation of the implicit terms and the linear
- *time_schemes* (type_tscheme()): This module defines an abstract class type_tscheme which is employed for the time advance of the code.
- *time_array* (type_tarray(), type_tscalar()): This module defines two types that are defined to store the implicit/explicit terms at the different sub-stage/step.

Variables

- readcheckpoints/**bytes_allocated** *[integer,private/optional/default=0]*
- readcheckpoints/**l_axi_old** *[logical,private]*
- readcheckpoints/**lreadr** *[logical,private]*
- readcheckpoints/**lreads** *[logical,private]*
- readcheckpoints/**lreadxi** *[logical,private]*
- readcheckpoints/**n_start_file** *[integer,private]*
- readcheckpoints/**ratio1** *[real,private]*
- readcheckpoints/**ratio1_old** *[real,private]*
- readcheckpoints/**ratio2** *[real,private]*
- readcheckpoints/**ratio2_old** *[real,private]*

Subroutines and functions

subroutine readcheckpoints/**readstartfields_old**(*w*, *dwdt*, *z*, *dzdt*, *p*, *dpdt*, *s*, *dsdt*, *xi*, *dxidt*, *phi*, *dphidt*, *b*, *dbdt*, *aj*, *djdt*, *b_ic*, *dbdt_ic*, *aj_ic*, *djdt_ic*, *omega_ic*, *omega_ma*, *domega_ic_dt*, *domega_ma_dt*, *lorentz_torque_ic_dt*, *lorentz_torque_ma_dt*, *time*, *tscheme*, *n_time_step*)

This subroutine is used to read the old restart files produced by MagIC. This is now deprecated with the change of the file format. This is still needed to read old files.

Parameters

- **w** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dwdt** *[type_tarray,inout]*
- **z** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dzdt** *[type_tarray,inout]*
- **p** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dpdt** *[type_tarray,inout]*
- **s** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dsdt** *[type_tarray,inout]*
- **xi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dxidt** *[type_tarray,inout]*
- **phi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dphidt** *[type_tarray,inout]*
- **b** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,out]*
- **dbdt** *[type_tarray,inout]*
- **aj** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,out]*
- **djdt** *[type_tarray,inout]*

- **b_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,out]*
- **dbdt_ic** *[type_tarray,inout]*
- **aj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,out]*
- **djdt_ic** *[type_tarray,inout]*
- **omega_ic** *[real,out]*
- **omega_ma** *[real,out]*
- **domega_ic_dt** *[type_tscalar,inout]*
- **domega_ma_dt** *[type_tscalar,inout]*
- **lorentz_torque_ic_dt** *[type_tscalar,inout]*
- **lorentz_torque_ma_dt** *[type_tscalar,inout]*
- **time** *[real,out]*
- **tscheme** *[real]*
- **n_time_step** *[integer,out]*

Called from *getstartfields()*

Call to *abortrun()*, *getlm2lmo()*, *mapdatahydro()*, *scatter_from_rank0_to_lo()*, *get_single_rhs_imp()*, *get_pol_rhs_imp()*, *get_entropy_rhs_imp()*, *get_tor_rhs_imp()*, *get_comp_rhs_imp()*, *mapdatamag()*, *get_mag_rhs_imp()*, *get_mag_ic_rhs_imp()*, *finish_start_fields()*

subroutine readcheckpoints/**readstartfields**(*w*, *dwdt*, *z*, *dzdt*, *p*, *dpdt*, *s*, *dsdt*, *xi*, *dxidt*, *phi*, *dphidt*, *b*, *dbdt*, *aj*, *djdt*, *b_ic*, *dbdt_ic*, *aj_ic*, *djdt_ic*, *omega_ic*, *omega_ma*, *domega_ic_dt*, *domega_ma_dt*, *lorentz_torque_ic_dt*, *lorentz_torque_ma_dt*, *time*, *tscheme*, *n_time_step*)

This subroutine is used to read the restart files produced by MagIC.

Parameters

- **w** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dwdt** *[type_tarray,inout]*
- **z** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dzdt** *[type_tarray,inout]*
- **p** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dpdt** *[type_tarray,inout]*
- **s** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dsdt** *[type_tarray,inout]*
- **xi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dxidt** *[type_tarray,inout]*
- **phi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dphidt** *[type_tarray,inout]*
- **b** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,out]*

- **dbdt** *[type_tarray,inout]*
- **aj** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,out]*
- **djdt** *[type_tarray,inout]*
- **b_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,out]*
- **dbdt_ic** *[type_tarray,inout]*
- **aj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,out]*
- **djdt_ic** *[type_tarray,inout]*
- **omega_ic** *[real,out]*
- **omega_ma** *[real,out]*
- **domega_ic_dt** *[type_tscalar,inout]*
- **domega_ma_dt** *[type_tscalar,inout]*
- **lorentz_torque_ic_dt** *[type_tscalar,inout]*
- **lorentz_torque_ma_dt** *[type_tscalar,inout]*
- **time** *[real,out]*
- **tscheme** *[real]*
- **n_time_step** *[integer,out]*

Called from *readstartfields_mpi()*

Call to *abortrun()*, *print_info()*, *getlm2lmo()*, *read_map_one_scalar()*, *read_map_one_field()*, *maponefield()*, *scatter_from_rank0_to_lo()*, *finish_start_fields()*, *get_single_rhs_imp()*, *get_pol_rhs_imp()*, *get_entropy_rhs_imp()*, *get_tor_rhs_imp()*, *get_comp_rhs_imp()*, *get_mag_rhs_imp()*, *get_mag_ic_rhs_imp()*

subroutine readcheckpoints/**read_map_one_scalar**(*fh*, *tscheme*, *nexp_old*, *nimp_old*, *nold_old*, *tscheme_family_old*, *dscal_dt*)

– Input variables

Parameters

- **fh** *[integer,in]*
- **tscheme** *[real]*
- **nexp_old** *[integer,in]*
- **nimp_old** *[integer,in]*
- **nold_old** *[integer,in]*
- **tscheme_family_old** *[character,in]*
- **dscal_dt** *[type_tscalar,inout]*

Called from *readstartfields()*

subroutine readcheckpoints/**read_map_one_field**(*fh*, *tscheme*, *wold*, *work*, *scale_w*, *r_old*, *lm2lmo*, *n_r_max_old*, *n_r_maxl*, *dim1*, *nexp_old*, *nimp_old*, *nold_old*, *tscheme_family_old*, *w*, *dwdt*, *l_map*, *l_transp*)

— Input variables

Parameters

- **fh** *[integer,in]*
- **tscheme** *[real]*
- **wold** (,) *[complex,inout]*
- **work** (,) *[complex,inout]*
- **scale_w** *[real,in]*
- **r_old** (*) *[real,in]*
- **lm2lmo** (*lm_max*) *[integer,in]*
- **n_r_max_old** *[integer,in]*
- **n_r_maxl** *[integer,in]*
- **dim1** *[integer,in]*
- **nexp_old** *[integer,in]*
- **nimp_old** *[integer,in]*
- **nold_old** *[integer,in]*
- **tscheme_family_old** *[character,in]*
- **w** (1 - *llm* + *ulm*,dim1) *[complex,out]*
- **dwdt** *[type_tarray,inout]*
- **l_map** *[logical,in]*
- **l_transp** *[logical,in]* :: do we need to transpose to lm-loc

Called from *readstartfields()*

Call to *maponefield()*, *scatter_from_rank0_to_lo()*

subroutine readcheckpoints/**readstartfields_mpi**(*w*, *dwdt*, *z*, *dzdt*, *p*, *dpdt*, *s*, *dsdt*, *xi*, *dxidt*, *phi*, *dphidt*, *b*, *dbdt*, *aj*, *djdt*, *b_ic*, *dbdt_ic*, *aj_ic*, *djdt_ic*, *omega_ic*, *omega_ma*, *domega_ic_dt*, *domega_ma_dt*, *lorentz_torque_ic_dt*, *lorentz_torque_ma_dt*, *time*, *tscheme*, *n_time_step*)

This subroutine is used to read the restart files produced by MagIC using MPI-IO

Parameters

- **w** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dwdt** *[type_tarray,inout]*
- **z** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dzdt** *[type_tarray,inout]*
- **p** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dpdt** *[type_tarray,inout]*
- **s** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dsdt** *[type_tarray,inout]*
- **xi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*

- **dxidt** *[type_tarray,inout]*
- **phi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,out]*
- **dphidt** *[type_tarray,inout]*
- **b** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,out]*
- **dbdt** *[type_tarray,inout]*
- **aj** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,out]*
- **djdt** *[type_tarray,inout]*
- **b_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,out]*
- **dbdt_ic** *[type_tarray,inout]*
- **aj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,out]*
- **djdt_ic** *[type_tarray,inout]*
- **omega_ic** *[real,out]*
- **omega_ma** *[real,out]*
- **domega_ic_dt** *[type_tscalar,inout]*
- **domega_ma_dt** *[type_tscalar,inout]*
- **lorentz_torque_ic_dt** *[type_tscalar,inout]*
- **lorentz_torque_ma_dt** *[type_tscalar,inout]*
- **time** *[real,out]*
- **tscheme** *[real]*
- **n_time_step** *[integer,out]*

Called from *getstartfields()*

Call to *mpiio_setup()*, *abortrun()*, *readstartfields()*, *print_info()*, *getlm2lmo()*, *read_map_one_scalar_mpi()*, *getblocks()*, *read_map_one_field_mpi()*, *maponefield()*, *scatter_from_rank0_to_lo()*, *get_single_rhs_imp()*, *get_pol_rhs_imp()*, *get_entropy_rhs_imp()*, *get_tor_rhs_imp()*, *get_comp_rhs_imp()*, *get_mag_rhs_imp()*, *get_mag_ic_rhs_imp()*, *finish_start_fields()*

subroutine readcheckpoints/**read_map_one_scalar_mpi**(*fh*, *tscheme*, *nexp_old*, *nimp_old*, *nold_old*, *tscheme_family_old*, *dscal_dt*)

– Input variables

Parameters

- **fh** *[integer,in]*
- **tscheme** *[real]*
- **nexp_old** *[integer,in]*
- **nimp_old** *[integer,in]*
- **nold_old** *[integer,in]*
- **tscheme_family_old** *[character,in]*
- **dscal_dt** *[type_tscalar,inout]*

Called from *readstartfields_mpi()*

subroutine readcheckpoints/**read_map_one_field_mpi**(*fh*, *info*, *datatype*, *tscheme*, *wold*, *lm_max_old*, *n_r_max_old*, *nrstart_old*, *nrstop_old*, *radial_balance_old*, *lm2lmo*, *r_old*, *n_r_maxl*, *dim1*, *scale_w*, *nexp_old*, *nimp_old*, *nold_old*, *tscheme_family_old*, *w*, *dwdt*, *disp*, *l_map*, *l_transp*)

— Input variables

Parameters

- **fh** *[integer,in]*
- **info** *[integer,in]*
- **datatype** *[integer,in]*
- **tscheme** *[real]*
- **wold** (lm_max_old,1 - nrstart_old + nrstop_old) *[complex,in]*
- **lm_max_old** *[integer,in,]*
- **n_r_max_old** *[integer,in]*
- **radial_balance_old** (*n_procs*) *[load,in]*
- **lm2lmo** (*lm_max*) *[integer,in]*
- **r_old** (*) *[real,in]*
- **n_r_maxl** *[integer,in]*
- **dim1** *[integer,in]*
- **scale_w** *[real,in]*
- **nexp_old** *[integer,in]*
- **nimp_old** *[integer,in]*
- **nold_old** *[integer,in]*
- **tscheme_family_old** *[character,in]*
- **w** (1 - *llm* + *ulm*,dim1) *[complex,out]*
- **dwdt** *[type_tarray,inout]*
- **disp** *[integer,inout]*
- **l_map** *[logical,in]*
- **l_transp** *[logical,in]* :: Do we need to transpose d?dt arrays?

Options

- **nrstart_old** *[integer,in,optional/default=(-1 - nrstop_old + shape(wold, 1)) / (-1)]*
- **nrstop_old** *[integer,in,optional/default=-1 + nrstart_old + shape(wold, 1)]*

Called from *readstartfields_mpi()*

Call to *maponefield_mpi()*

subroutine readcheckpoints/**getlm2lmo**(*n_r_max*, *n_r_max_old*, *l_max*, *l_max_old*, *m_max*, *minc*, *minc_old*, *lm_max*, *lm_max_old*, *lm2lmo*)

— Input variables

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]*
- **lm_max** *[integer,in]*
- **lm_max_old** *[integer,out]*
- **lm2lmo** (*lm_max*) *[integer,out]* :: data found in startfile

Called from *readstartfields_old()*, *readstartfields()*, *readstartfields_mpi()*

subroutine readcheckpoints/**maponefield_mpi**(*wold*, *lm_max_old*, *n_r_max_old*, *nrstart_old*, *nrstop_old*, *radial_balance_old*, *lm2lmo*, *r_old*, *n_r_maxl*, *dim1*, *lbc1*, *l_ic*, *scale_w*, *w*)

— Input variables

Parameters

- **wold** (lm_max_old,1 - nrstart_old + nrstop_old) *[complex,in]*
- **lm_max_old** *[integer,in,]*
- **n_r_max_old** *[integer,in]*
- **radial_balance_old** (*n_procs*) *[load,in]*
- **lm2lmo** (*lm_max*) *[integer,in]*
- **r_old** (*) *[real,in]*
- **n_r_maxl** *[integer,in]*
- **dim1** *[integer,in]*
- **lbc1** *[logical,in]*
- **l_ic** *[logical,in]*
- **scale_w** *[real,in]*
- **w** (1 - *llm* + *ulm*,dim1) *[complex,out]*

Options

- **nrstart_old** *[integer,in,optional/default=(-1 - nrstop_old + shape(wold, 1)) / (-1)]*
- **nrstop_old** *[integer,in,optional/default=-1 + nrstart_old + shape(wold, 1)]*

Called from *read_map_one_field_mpi()*

Call to *mapdatar()*

subroutine readcheckpoints/**maponefield**(*wo*, *scale_w*, *r_old*, *lm2lmo*, *n_r_max_old*, *n_r_maxl*, *dim1*, *lbc1*, *l_ic*, *w*)

— Input variables

Parameters

- **wo** (,) *[complex,in]*
- **scale_w** *[real,in]*
- **r_old** (*) *[real,in]*
- **lm2lmo** (*lm_max*) *[integer,in]*
- **n_r_max_old** *[integer,in]*
- **n_r_maxl** *[integer,in]*
- **dim1** *[integer,in]*
- **lbc1** *[logical,in]*
- **l_ic** *[logical,in]*
- **w** (*lm_max*,dim1) *[complex,out]*

Called from *readstartfields()*, *read_map_one_field()*, *readstartfields_mpi()*

Call to *mapdatar()*

subroutine readcheckpoints/**mapdatahydro**(*wo*, *zo*, *po*, *so*, *xio*, *r_old*, *lm2lmo*, *n_r_max_old*, *lm_max_old*, *n_r_maxl*, *lbc1*, *lbc2*, *lbc3*, *lbc4*, *lbc5*, *w*, *z*, *p*, *s*, *xi*)

— Input variables

Parameters

- **wo** (lm_max_old,n_r_max_old) *[complex,in]*
- **zo** (lm_max_old,n_r_max_old) *[complex,in]*
- **po** (lm_max_old,n_r_max_old) *[complex,in]*
- **so** (lm_max_old,n_r_max_old) *[complex,in]*
- **xio** (lm_max_old,n_r_max_old) *[complex,in]*
- **r_old** (*) *[real,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]*
- **lbc5** *[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]*
- **xi** (*lm_max*,*n_r_max*) *[complex,out]*

Called from *readstartfields_old()*

Call to *mapdatar()*

subroutine readcheckpoints/**mapdatamag**(*wo*, *zo*, *po*, *so*, *r_old*, *n_rad_tot*, *n_r_max_old*, *lm_max_old*, *n_r_maxl*, *lm2lmo*, *dim1*, *l_ic*, *w*, *z*, *p*, *s*)

— Input variables

Parameters

- **wo** (lm_max_old,n_r_max_old) *[complex,in]*
- **zo** (lm_max_old,n_r_max_old) *[complex,in]*
- **po** (lm_max_old,n_r_max_old) *[complex,in]*
- **so** (lm_max_old,n_r_max_old) *[complex,in]*
- **r_old** (*) *[real,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_old()*

Call to *mapdatar()*

subroutine readcheckpoints/**mapdatar**(*datar*, *r_old*, *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,inout]* :: old data
- **r_old** (*) *[real,in]*
- **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 *maponefield_mpi()*, *maponefield()*, *mapdatahydro()*, *mapdatamag()*

Call to *polynomial_interpolation()*

subroutine readcheckpoints/**finish_start_fields**(*time*, *minc_old*, *l_mag_old*, *omega_ic1old*, *omega_ma1old*, *z*, *s*, *xi*, *b*, *omega_ic*, *omega_ma*)

– Input variables

Parameters

- **time** *[real,in]*
- **minc_old** *[integer,in]*
- **l_mag_old** *[logical,in]*
- **omega_ic1old** *[real,in]*
- **omega_ma1old** *[real,in]*
- **z** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **s** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **xi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **b** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,inout]*
- **omega_ic** *[real,out]*
- **omega_ma** *[real,out]*

Called from *readstartfields_old()*, *readstartfields()*, *readstartfields_mpi()*

subroutine readcheckpoints/**print_info**(*ra_old*, *ek_old*, *pr_old*, *sc_old*, *raxi_old*, *pm_old*, *radratio_old*, *sigma_ratio_old*, *n_phi_tot_old*, *nalias_old*, *l_max_old*)

– Input variables

Parameters

- **ra_old** *[real,in]*
- **ek_old** *[real,in]*
- **pr_old** *[real,in]*
- **sc_old** *[real,in]*
- **raxi_old** *[real,in]*
- **pm_old** *[real,in]*
- **radratio_old** *[real,in]*
- **sigma_ratio_old** *[real,in]*
- **n_phi_tot_old** *[integer,in]*
- **nalias_old** *[integer,in]*
- **l_max_old** *[integer,in]*

10.20.2 storeCheckPoints.f90

Description

This module contains several subroutines that can be used to store the checkpoint_#.tag files

Quick access

Routines *store()*, *store_mpi()*, *write_one_field()*, *write_one_field_mpi()*

Needed modules

- iso_fortran_env (output_unit())
- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod*: This module contains the blocking information
- *communications* (*gt_oc()*, *gt_ic()*, *gather_from_lo_to_rank0()*, *gather_all_from_lo_to_rank0()*, lo2r_one()): This module contains the different MPI communicators used in MagIC.
- *truncation* (*n_r_max()*, *n_r_ic_max()*, *minc()*, *nalias()*, *n_theta_max()*, *n_phi_tot()*, *lm_max()*, *lm_maxmag()*, *n_r_maxmag()*, *n_r_ic_maxmag()*, *l_max()*, *fd_stretch()*, *fd_ratio()*): This module defines the grid points and the truncation
- *radial_functions* (rscheme_oc(), *r()*): This module initiates all the radial functions (transport properties, density, temperature, cheb transforms, etc.)
- *physical_parameters* (*ra()*, *pr()*, *prmag()*, *radratio()*, *ek()*, *sigma_ratio()*, *raxi()*, *sc()*, *stef()*): Module containing the physical parameters
- *blocking* (*llm()*, *ulm()*, *llmmag()*, *ulmmag()*): Module containing blocking information
- *radial_data* (*nrstart()*, *nrstop()*, *nrstartmag()*, *nrstopmag()*): This module defines the MPI decomposition in the radial direction.
- *num_param* (*tscale()*, *alph1()*, *alph2()*): Module containing numerical and control parameters
- *init_fields* (*inform()*, *omega_ic1()*, *omegaosz_ic1()*, *tomega_ic1()*, *omega_ic2()*, *omegaosz_ic2()*, *tomega_ic2()*, *omega_ma1()*, *omegaosz_ma1()*, *tomega_ma1()*, *omega_ma2()*, *omegaosz_ma2()*, *tomega_ma2()*): This module is used to construct the initial solution.
- *logic* (*l_heat()*, *l_mag()*, *l_cond_ic()*, *l_chemical_conv()*, *l_save_out()*, *l_double_curl()*, *l_parallel_solve()*, *l_mag_par_solve()*, *l_phase_field()*): Module containing the logicals that control the run
- *output_data* (*tag()*, *log_file()*, *n_log_file()*): This module contains the parameters for output control
- *charmanip* (*dble2str()*): This module contains several useful routines to manipule character strings
- *time_schemes* (type_tscheme()): This module defines an abstract class type_tscheme which is employed for the time advance of the code.
- *time_array* (type_tarray(), type_tscalar()): This module defines two types that are defined to store the implicit/explicit terms at the different sub-stage/step.

Variables

Subroutines and functions

subroutine storecheckpoints/**store**(*time*, *tscheme*, *n_time_step*, *l_stop_time*, *l_new_rst_file*, *l_ave_file*, *w*, *z*, *p*, *s*, *xi*, *phi*, *b*, *aj*, *b_ic*, *aj_ic*, *dwdt*, *dzdt*, *dpdt*, *dsdt*, *dxidt*, *dphidt*, *dbdt*, *djdt*, *dbdt_ic*, *djdt_ic*, *domega_ma_dt*, *domega_ic_dt*, *lorentz_torque_ma_dt*, *lorentz_torque_ic_dt*)

This subroutine stores the results in a checkpoint file. In addition to the magnetic field and velocity potentials we also store the time derivative terms djdt(lm,nR),dbdt(lm,nR), ... to allow to restart with 2nd order Adams-Bashforth scheme. To minimize the memory imprint, a gather/write strategy has been adopted here. This implies that only one global array dimension(lm_max,n_r_max) is required.

Parameters

- **time** *[real,in]*
- **tscheme** *[real]*
- **n_time_step** *[integer,in]*
- **l_stop_time** *[logical,in]*
- **l_new_rst_file** *[logical,in]*
- **l_ave_file** *[logical,in]*
- **w** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **z** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **p** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **s** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **xi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **phi** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **b** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]*
- **aj** (1 - *llmmag* + *ulmmag*,*n_r_maxmag*) *[complex,in]*
- **b_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,in]*
- **aj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,in]*
- **dwdt** *[type_tarray,in]*
- **dzdt** *[type_tarray,in]*
- **dpdt** *[type_tarray,in]*
- **dsdt** *[type_tarray,in]*
- **dxidt** *[type_tarray,in]*
- **dphidt** *[type_tarray,in]*
- **dbdt** *[type_tarray,in]*
- **djdt** *[type_tarray,in]*
- **dbdt_ic** *[type_tarray,in]*
- **djdt_ic** *[type_tarray,in]*

- **domega_ma_dt** *[type_tscalar,in]*
- **domega_ic_dt** *[type_tscalar,in]*
- **lorentz_torque_ma_dt** *[type_tscalar,in]*
- **lorentz_torque_ic_dt** *[type_tscalar,in]*

Called from *fields_average()*

Call to *dble2str()*, *write_one_field()*, *gather_all_from_lo_to_rank0()*

subroutine storecheckpoints/**write_one_field**(*fh*, *tscheme*, *w*, *dwdt*, *work*)

Parameters

- **fh** *[integer,in]* :: file unit
- **tscheme** *[real]* :: Time scheme
- **w** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]* :: field
- **dwdt** *[type_tarray,in]*
- **work** (,) *[complex,inout]*

Called from *graphout_mpi()*, *graphout_ic()*, *store()*

Call to *gather_all_from_lo_to_rank0()*

subroutine storecheckpoints/**store_mpi**(*time*, *tscheme*, *n_time_step*, *l_stop_time*, *l_new_rst_file*, *l_ave_file*, *w*, *z*, *p*, *s*, *xi*, *phi*, *b*, *aj*, *b_ic*, *aj_ic*, *dwdt*, *dzdt*, *dpdt*, *dsdt*, *dxidt*, *dphidt*, *dbdt*, *djdt*, *dbdt_ic*, *djdt_ic*, *domega_ma_dt*, *domega_ic_dt*, *lorentz_torque_ma_dt*, *lorentz_torque_ic_dt*)

This subroutine stores the results in a checkpoint file. In addition to the magnetic field and velocity potentials we also store the time derivative terms djdt(lm,nR),dbdt(lm,nR), ... to allow to restart with 2nd order Adams-Bashforth scheme. To minimize the memory imprint, a gather/write strategy has been adopted here. This implies that only one global array dimension(lm_max,n_r_max) is required.

Parameters

- **time** *[real,in]*
- **tscheme** *[real]*
- **n_time_step** *[integer,in]*
- **l_stop_time** *[logical,in]*
- **l_new_rst_file** *[logical,in]*
- **l_ave_file** *[logical,in]*
- **w** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,in]*
- **z** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,in]*
- **p** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,in]*
- **s** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,in]*
- **xi** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,in]*
- **phi** (*lm_max*,1 - *nrstart* + *nrstop*) *[complex,in]*

- **b** (*lm_maxmag*,1 - *nrstartmag* + *nrstopmag*) *[complex,in]*
- **aj** (*lm_maxmag*,1 - *nrstartmag* + *nrstopmag*) *[complex,in]*
- **b_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,in]*
- **aj_ic** (1 - *llmmag* + *ulmmag*,*n_r_ic_maxmag*) *[complex,in]*
- **dwdt** *[type_tarray,in]*
- **dzdt** *[type_tarray,in]*
- **dpdt** *[type_tarray,in]*
- **dsdt** *[type_tarray,in]*
- **dxidt** *[type_tarray,in]*
- **dphidt** *[type_tarray,in]*
- **dbdt** *[type_tarray,in]*
- **djdt** *[type_tarray,in]*
- **dbdt_ic** *[type_tarray,in]*
- **djdt_ic** *[type_tarray,in]*
- **domega_ma_dt** *[type_tscalar,in]*
- **domega_ic_dt** *[type_tscalar,in]*
- **lorentz_torque_ma_dt** *[type_tscalar,in]*
- **lorentz_torque_ic_dt** *[type_tscalar,in]*

Called from *output()*

Call to *dble2str()*, *mpiio_setup()*, *write_one_field_mpi()*, *gather_from_lo_to_rank0()*

subroutine storecheckpoints/**write_one_field_mpi**(*fh*, *info*, *datatype*, *tscheme*, *w*, *dwdt*, *work*, *size_tmp*, *disp*, *l_transp*)

This subroutine is used to write one field and its associated possible help arrays (d?dt) which are required to time advance the solution. This is using MPI-IO with R-distributed arrays.

Parameters

- **fh** *[integer,in]* :: file unit
- **info** *[integer,in]* :: MPI file info
- **datatype** *[integer,in]* :: MPI file info
- **tscheme** *[real]*
- **w** (1 - *llm* + *ulm*,*n_r_max*) *[complex,in]*
- **dwdt** *[type_tarray,in]* :: time advance arrays
- **work** (1 - *llm* + *ulm*,*n_r_max*) *[complex,inout]*
- **size_tmp** *[integer,in]*
- **disp** *[integer,inout]*
- **l_transp** *[logical,in]* :: Do we need to transpose anything?

Called from *store_mpi()*

10.21 Useful additional libraries

10.21.1 mean_sd.f90

Description

This module contains a small type that simply handles two arrays (mean and SD) This type is used for time-averaged outputs (and their standard deviations).

Quick access

Routines *compute_1d()*, *compute_2d_1d_input()*, *compute_2d_2d_input()*, *finalize_1d()*, *finalize_2d()*, *finalize_sd_1d()*, *finalize_sd_2d()*, *initialize_1d()*, *initialize_2d()*

Needed modules

- *mem_alloc*: This little module is used to estimate the global memory allocation used in MagIC
- *precision_mod*: This module controls the precision used in MagIC

Types

- **type** mean_sd/**unknown_type**

  Type fields

  - **% mean** (*) *[real,allocatable]*
  - **% sd** (*) *[real,allocatable]*

- **type** mean_sd/**unknown_type**

  Type fields

  - **% l_sd** *[logical]*
  - **% mean** (,) *[real,allocatable]*
  - **% sd** (,) *[real,allocatable]*

Subroutines and functions

subroutine mean_sd/**initialize_1d**(*this*, *n_start*, *n_stop*)

Memory allocation

Parameters

- **this** *[real]*
- **n_start** *[integer,in]*
- **n_stop** *[integer,in]*

subroutine mean_sd/**compute_1d**(*this*, *input_data*, *n_ave*, *dt*, *totaltime*)

Parameters

- **this** *[real]*
- **input_data** (*) *[real,in]*
- **n_ave** *[integer,in]*
- **dt** *[real,in]*
- **totaltime** *[real,in]*

subroutine mean_sd/**finalize_sd_1d**(*this*, *totaltime*)

Finish computation of standard-deviation

Parameters

- **this** *[real]*
- **totaltime** *[real,in]*

subroutine mean_sd/**finalize_1d**(*this*)

Memory deallocation

Parameters **this** *[real]*

subroutine mean_sd/**initialize_2d**(*this*, *n_start_row*, *n_stop_row*, *n_start_col*, *n_stop_col*, *l_sd*)

Memory allocation

Parameters

- **this** *[real]*
- **n_start_row** *[integer,in]*
- **n_stop_row** *[integer,in]*
- **n_start_col** *[integer,in]*
- **n_stop_col** *[integer,in]*
- **l_sd** *[logical,in]*

subroutine mean_sd/**compute_2d_1d_input**(*this*, *input_data*, *n_ave*, *dt*, *totaltime*, *ind*)

Parameters

- **this** *[real]*
- **input_data** (*) *[real,in]*
- **n_ave** *[integer,in]*
- **dt** *[real,in]*
- **totaltime** *[real,in]*
- **ind** *[integer,in]*

subroutine mean_sd/**compute_2d_2d_input**(*this*, *input_data*, *n_ave*, *dt*, *totaltime*)

Parameters

- **this** *[real]*
- **input_data** (,) *[real,in]*
- **n_ave** *[integer,in]*
- **dt** *[real,in]*
- **totaltime** *[real,in]*

subroutine mean_sd/**finalize_sd_2d**(*this*, *totaltime*)

Finish computation of standard-deviation

Parameters

- **this** *[real]*
- **totaltime** *[real,in]*

subroutine mean_sd/**finalize_2d**(*this*)

Memory deallocation

Parameters **this** *[real]*

10.21.2 useful.f90

Description

This module contains several useful routines.

Quick access

Routines *abortrun()*, *cc22real()*, *cc2real()*, *factorise()*, *l_correct_step()*, *logwrite()*, *polynomial_interpolation()*, *polynomial_interpolation_real()*, *random()*, *round_off()*

Needed modules

- `iso_fortran_env` (`output_unit()`)
- *precision_mod*: This module controls the precision used in MagIC
- *parallel_mod*: This module contains the blocking information
- *output_data* (*n_log_file()*, *log_file()*): This module contains the parameters for output control
- *logic* (*l_save_out()*): Module containing the logicals that control the run
- *constants* (*half()*, *one()*, *two()*): module containing constants and parameters used in the code.

Variables

Subroutines and functions

function useful/**l_correct_step**(*n*, *t*, *t_last*, *n_max*, *n_step*, *n_intervals*, *n_ts*, *times*, *n_eo*)

Suppose we have a (loop) maximum of n_max steps! If n_intervals times in these steps a certain action should be carried out this can be invoked by l_correct_step=true if on input n_intervals>0 and n_step=0. Alternatively the action can be invoked every n_step steps if on input n_intervals=0 and n_step>0. In both cases l_correct_step=true for n=n_max.

The argument controls whether in addition n should be even (n_eo=2) or odd (n_eo=1)

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]* :: ’
- **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()*

Call to *abortrun()*

function useful/**random**(*r*)

Random number generator:

- if (r == 0) random(r) = next random number (between 0. and 1.)
- if (r < 0) random(r) = previous random number
- if (r > 0) random(r) = a new sequence of random numbers is started with seed r mod 1

Parameters **r** *[real,in]*

Return **random** *[real]*

Called from *initv()*, *inits()*, *initxi()*, *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]* :: remaining
- **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
- **factor** (*) *[integer,out]*

Called from *init_fft()*

Call to *abortrun()*

function useful/**cc2real**(*c*, *m*)

This function computes the norm of complex number, depending on the azimuthal wavenumber.

Parameters

- **c** *[complex,in]* :: A complex number
- **m** *[integer,in]* :: Azimuthal wavenumber

Return **cc2real** *[real]*

Called from *get_poltorrms()*, *hint2dpol()*, *hint2dpollm()*, *hint2pol()*, *hint2pollm()*, *hintrms()*, *hint2tor()*, *hint2torlm()*, *getdlm()*, *get_e_kin()*, *get_u_square()*, *get_e_mag()*, *get_power()*, *rbrspec()*, *rbpspec()*, *spectrum()*, *spectrum_temp()*, *get_amplitude()*, *getstartfields()*

function useful/**cc22real**(*c1*, *c2*, *m*)

Parameters

- **c1** *[complex,in]*
- **c2** *[complex,in]*
- **m** *[integer,in]*

Return **cc22real** *[real]*

Called from *get_e_mag()*, *spectrum()*

subroutine useful/**logwrite**(*message*)

This subroutine writes a message in the log.TAG file and in the STDOUT If l_save_out is set to .true. the log.TAG file is opened and closed.

Parameters **message** *[character,in]* :: Message to be written

Called from *dt_courant()*, *get_fd_grid()*, *outto()*, *output()*, *precalc()*, *radial()*, *getstartfields()*, *step_time()*

subroutine useful/**polynomial_interpolation**(*xold*, *yold*, *xnew*, *ynew*)

Parameters

- **xold** (4) *[real,in]*
- **yold** (4) *[complex,in]*
- **xnew** *[real,in]*
- **ynew** *[complex,out]*

Called from *mapdatar()*

Call to *polynomial_interpolation_real()*

subroutine useful/**polynomial_interpolation_real**(*xold*, *yold*, *xnew*, *ynew*)

This subroutine performs a polynomial interpolation of ynew(xnew) using input data xold and yold.

Parameters

- **xold** (*) *[real,in]* :: Old grid
- **yold** (*) *[real,in]* :: Old data
- **xnew** *[real,in]* :: Point of interpolation
- **ynew** *[real,out]* :: Data interpolated at xnew

Called from *polynomial_interpolation()*

Call to *abortrun()*

subroutine useful/**abortrun**(*message*)

This routine properly terminates a run

Parameters **message** *[character,in]* :: Message printing before termination

Called from *parallel_solve_phase()*, *parallel_solve()*, *readnamelists()*, *init_rnb()*, *prepare_mat()*, *initialize_blocking()*, *get_subblocks()*, *get_standard_lm_blocking()*, *get_lorder_lm_blocking()*, *get_snake_lm_blocking()*, *lm2lo_redist()*, *lo2lm_redist()*, *set_dt_array()*, *init_fft()*, *get_fd_grid()*, *getdlm()*, *inits()*, *initxi()*, *initb()*, *j_cond()*, *xi_cond()*, *pt_cond()*, *ps_cond()*, *get_movie_type()*, *movie_gather_frames_to_rank0()*, *get_b_nl_bcs()*, *write_movie_frame()*, *plm_theta()*, *precalc()*, *get_hit_times()*, *writeinfo()*, *transportproperties()*, *getbackground()*, *get_dr_rloc()*, *get_ddr_rloc()*, *get_ddr_ghost()*, *get_ddddr_ghost()*, *readstartfields_old()*, *readstartfields()*, *readstartfields_mpi()*, *checktruncation()*, *updateb()*, *prepareb_fd()*,

fill_ghosts_b(), *assemble_mag()*, *assemble_mag_rloc()*, *get_bmat()*, *get_bmat_rdist()*, *get_phi0mat()*, *get_phimat()*, *get_s0mat()*, *get_smat()*, *assemble_pol_rloc()*, *get_wpmat()*, *get_elliptic_mat()*, *get_wmat()*, *get_p0mat()*, *assemble_single()*, *get_wpsmat()*, *get_ps0mat()*, *get_xi0mat()*, *get_ximat()*, *assemble_tor()*, *assemble_tor_rloc()*, *get_z10mat()*, *get_zmat()*, *get_z10mat_rdist()*, *l_correct_step()*, *factorise()*, *polynomial_interpolation_real()*

function useful/**round_off**(*param*, *ref*)

This function rounds off tiny numbers. This is only used for some outputs.

Parameters

- **param** *[real,in]* :: parameter to be checked
- **ref** *[real,in]* :: reference value

Return **round_off** *[real]*

Called from *outheat()*, *outphase()*, *outpar()*, *outperppar()*, *get_power()*

10.21.3 mem_alloc.f90

Description

This little module is used to estimate the global memory allocation used in MagIC

Quick access

Variables *memory_file*, *n_memory_file*, *n_ranks_print*, *ranks_selected*

Routines *finalize_memory_counter()*, *human_readable_size()*, *initialize_memory_counter()*, *memwrite()*

Needed modules

- *parallel_mod*: This module contains the blocking information
- *precision_mod* (*lip()*, *cp()*): This module controls the precision used in MagIC
- *output_data* (*tag()*): This module contains the parameters for output control

Variables

- mem_alloc/**bytes_allocated** *[integer,public]*
- mem_alloc/**memory_file** *[character,private]*
- mem_alloc/**n_memory_file** *[integer,private]*
- mem_alloc/**n_ranks_print** *[integer,private]*
- mem_alloc/**ranks_selected** (*) *[integer,private/allocatable]*

Subroutines and functions

subroutine mem_alloc/**initialize_memory_counter**()

Called from *magic*

subroutine mem_alloc/**memwrite**(*origin*, *bytes_alloc*)

Parameters

- **origin** *[character,in]*
- **bytes_alloc** *[integer,in]*

Called from *initialize_lmloop()*, *initialize_blocking()*, *initialize_communications()*, *initialize_radialloop()*, *magic*

Call to *human_readable_size()*

subroutine mem_alloc/**finalize_memory_counter**()

Called from *magic*

Call to *human_readable_size()*

function mem_alloc/**human_readable_size**(*bytes*)

Parameters **bytes** *[integer,in]*

Return **st** *[real]*

Called from *memwrite()*, *finalize_memory_counter()*

10.21.4 char_manip.f90

Description

This module contains several useful routines to manipule character strings

Quick access

Routines *capitalize()*, *dble2str()*, *delete_string()*, *length_to_blank()*, *length_to_char()*, *str2dble()*, *write_long_string()*

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 *readnamelists()*, *select_tscheme()*, *initialize_communications()*, *get_movie_type()*, read_signal_file()

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 *readnamelists()*, *writenamelists()*, *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]* :: char not found !

subroutine charmanip/**dble2str**(*num*, *str*)

converts a 1 number num into a character str

Parameters

- **num** *[real,in]*
- **str** *[character,out]*

Called from *get_movie_type()*, *store()*, *store_mpi()*

subroutine charmanip/**write_long_string**(*prefix*, *long_string*, *out_unit*)

This subroutine is used to split a long string (with len(str)>85) into a multi-lines string for a cleaner printout.

Parameters

- **prefix** *[character,in]*
- **long_string** *[character,in]*
- **out_unit** *[integer,in]*

Called from *magic*

PYTHON MODULE INDEX

m

FORTRAN MODULE INDEX

r

s

t

u

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z