ESTER

Evolution STEllaire en Rotation

User's Guide

January 13, 2015

Contents

1 Getting started

1.1. Prerequisites

The ESTER library depends on some external libraries that should be installed in the system, namely:

- BLAS, CBLAS and LAPACK, for matrix algebra. There are several versions available, as for example:
  - Netlib. This is the original implementation. The LAPACK library can be found at `http://www.netlib.org/lapack`, and already contains BLAS, but CBLAS should be downloaded separately from `http://www.netlib.org/blas`.
  - ATLAS (Automatically Tuned Linear Algebra Software). An implementation of LAPACK/BLAS that is automatically optimized during the compilation process. It can be found at `http://math-atlas.sourceforge.net/`. It contains LAPACK, BLAS and CBLAS.
  - Intel MKL. Contains an optimized version of LAPACK, BLAS and CBLAS for Intel processors.
- PGPLOT (CPGPLOT) for graphics output (optional). It can be disabled in the `Makefile` (`make.inc`) setting the variable `USE_PGPLOT=0`. This software is freely available on the web as, for instance, in the pgplot5 package of Ubuntu 10.04, 12.04, 14.04.

As there are some routines written in Fortran, it is also needed to link against the standard fortran libraries (`libgfortran` for the GNU fortran compiler and `libifcore` and `libifport` for the Intel compiler).

1.1.1. A note about the performance of the code

The performance of the ESTER code depends strongly on LAPACK. To get the best results, use an optimized (and parallelized) version.

1.2. Installation

The current version of the ESTER code can be downloaded using `svn` from the project server by doing

```
$ svn checkout http://ester-project.googlecode.com/svn/trunk/ ester
```

or from the project website http://code.google.com/p/ester-project.

The first step is to create the file `make.inc` in the directory `src` from the two examples that are included, `make.inc.icc` and `make.inc.gcc`, for the Intel compiler and the GNU compiler respectively.

For instance, if you have the Intel compilers (ifort, icpc), just do

```
ester/src$ cp make.inc.icc make.inc
```

```
ester/src$ make tables
```

This will build some third-party libraries included in the distribution and initialise the tables of opacity and equation of state.

You can now build the main program by doing

```
ester/src$ make
```

To remove intermediate files

```
ester/src$ make clean
```

Finally, to verify the installation, just

```
ester/src$ make test
```

These tests compute a 5 $M_\odot$ 1D model, plus a 5 $M_\odot$ 2D model with $\omega = 0.5$, then a 10 $M_\odot$ 1D model, plus a 10 $M_\odot$ 2D model with $\omega = 0.3$.

The main executable is located in `ester/bin/`. To be able to call ester without including the full path, you can include this directory in your PATH environment variable. Alternatively, you can create a symbolic link in a directory included in your PATH, for example:

```
$ ln -s ~/ester/bin/ester ~/bin/ester
```

In this example, we are assuming that the ESTER library is located in `~/ester` and the directory `~/bin` is included in the PATH. If you are interested also in making your own programs using the ESTER library, you can also do:

```
$ ln -s ~/ester/bin/ester_build ~/bin/ester_build
```

1.2.1. Updating the code

In order to update to the last version using `svn`, from the root directory of the ESTER distribution execute

```
ester$ svn update
```

Depending on the update, sometimes we can do just

```
ester/src$ make
```

from the `src` directory. But it is safer to clean out the previous installation using

```
ester/src$ make distclean
```

and then

```
ester/src$ make tables; make
```

1.3. Checking the installation

To check the functionality of the program we are going to calculate the structure of a star using the default values for the parameters. First we calculate the structure of the corresponding 1D non-rotating star. Change to your working directory and execute

```
$ ester 1d
```

Then we use the output file (by default `star.out`) as the starting point for the 2D calculation

```
$ ester 2d -i star.out -Omega_bk 0.5
```

This calculates the structure of a star rotating at 50% of the break-up velocity $\Omega_k = \sqrt{\frac{GM}{R_e^3}}$.

1.4. Using the library

The ESTER code can be used as a C++ library. We just have to add the following line at the beginning of our C++ program

```
#include "ester.h"
```

The main library is created in `ester/lib/libester.so` and the header files are in `ester/include`. To facilitate the process of compiling and linking against the library and all its dependencies, we provide an automatically generated script `ester/bin/ester_build` so, all you have to do is

```
$ ester_build your_cpp_program.cpp -o your_executable
```

2 Basic usage

The ESTER code calculates the structure and mean flows of an isolated (non-magnetic) rotating star. It uses realistic physics (tabulated opacity and EOS) and completely accounts for the deformation of the star. The mean flows are calculated self-consistently in the limit of low viscosity, so there is no need to impose an arbitrary prescription for the differential rotation of the star.

At this moment, chemical evolution is not included, but it can be faked by tweaking the fractional abundance of hydrogen in the convective core.

Surface convection is not included yet, so the computation is limited to stars with mass above $\sim 2M_{\odot}$.

For a detailed description of the physics involved in the models, see Rieutord & Espinosa Lara 2012 (arXiv:1208.4926) and Espinosa Lara & Rieutord 2013 (arXiv:1212.0778, submitted to A&A). The code is still in development, so new functionality will be added in future versions.

To execute the program, use the following syntax:

ester *command* [*options*]

where *command* can be

- `1d` Calculate the structure of a 1D non-rotating star
- `2d` Calculate the structure of a 2D rotating star
- `output` Generate a custom output file
- `info` Get information about a model file
- `help` Get help

2.1. Configuration files

The main configuration file is located at `ester/config/star.cfg`. This file contains the main options for the program, which are

- `maxit` (default 200). Maximum number of iterations. After `maxit` iterations, the program exits normally and the output file is saved, even if it has not completely converged.

- `minit` (default 1). Minimum number of iterations. It may occur that the value of the error for the first iteration is not representative. With this parameter we force the solver to do at least `minit` iterations. This parameter is superseded by `maxit`, for example if `maxit=5` and `minit=10`, the solver will do only 5 iterations.
- `tol` (default 1e-8). The relative tolerance for checking the convergence of the model.
- `newton_dmax` (default 0.5). After one step of the Newton's method, the maximum relative change allowed for a variable is given by `newton_dmax`. If necessary the iteration is relaxed by a parameter h

  $$\vec{x}^{N+1} = \vec{x}^N + h\delta\vec{x}^N$$

  according to this value. This parameter can be used to stabilize the convergence when the initial estimation is far from the solution.
- `output_file` (default `star.out`). Name of the output file.
- `output_mode` (default `b`). Type of the output file `b` for binary and `t` for text output.
- `verbose` (default 1). Level of verbosity, from 0 (quiet) to 4.
- `plot_device` (default `/XSERVE`. Plotting device for PGPLOT, see the documentation of PGPLOT for details. For output in a X window use `/XSERVE`. To disable the graphic output use `/NULL`.
- `plot_interval` (default 10). Minimum time in seconds to update the graphic output.

All this options can be specified in the file `ester/config/star.cfg` in the form `option_name=option_value` (one per line) and in the command line as `-option_name option_value`. The options specified in the command line have precedence over those specified in the configuration file.

There are some additional options that can be included in the command line:

`-input_file` *infile*. Use the file *infile* as the starting point for the iteration.

`-i` *infile*. Same as `-input_file` *infile*.

`-o` *outfile*. Same as `-output_file` *outfile*.

`-param_file` *file*. Where *file* contains the parameters of the stellar model to be calculated (see below).

`-p` *file*. Same as `-param_file` *file*.

`-ascii`. Same as `-output_mode t`.

`-binary`. Same as `-output_mode b`.

`-noplot`. Same as `-plot_device /NULL`.

`-v`*n*. Same as `-verbose` *n*.

2.2. Default values

Default values to be used by `star1d` or `star2d` may be set up with the files `ester/config/1d_default.par` and `ester/config/2d_default.par`.

In the distribution of ESTER, the proposed default values are such that the star is divided in 8 domains with 30 points in each domains. Opacities and equation of state are computed through OPAL tables. These inputs allow the calculation of a 3 $M_\odot$ model (but not only of course) from scratch.

2.3. `ester 1d` input parameters

The input parameters for `ester 1d` can be passed in the command line or in a text file specified with the option `-param_file` *file* (or just `-p` *file*). It can also be used simultaneously, in this case the parameters given in the command line take precedence over those specified in the file. In the text file they are written in the form `param_name`=*param_value* and in the command line as `-param_name` *param_value*. Here is the list of valid parameters

- `ndomains`. The number of subdomains to use.
- `npts`. Number of points in each subdomain. It is specified as a comma-separated list. If only one value is specified, it will be used for all the subdomains, for example:

  ```
  $ star1d -ndomains 4 -npts 20,20,20,20
  ```

  is equivalent to

  ```
  $ star1d -ndomains 4 -npts 20
  ```

- `M`. The mass in units of solar mass.
- `X`. Mass fraction of hydrogen.
- `Z`. Mass fraction of metals.
- `Xc`. Fraction of the hydrogen abundance present in the convective core. The profile of hydrogen abundance will be in the form

  $$X(\vec{r}) = \begin{cases} \mathtt{X} \times \mathtt{Xc} & \text{if } \vec{r} \text{ is in the convective core} \\ \mathtt{X} & \text{otherwise} \end{cases}$$

  If there is no convective core, this parameter is ignored.

- `surff`. This parameter is used for truncating the stellar model at some point below the surface. The surface pressure will be `surff` times the "real value and the boundary conditions will be adjusted in consequence. This parameter is provided only for testing purposes as it does not produce an accurate representation of the internal layers of the star. For regular calculations it should be `surff=1`.
- `Tc`. Initial estimation of the central temperature. To be updated during the calculation.
- `pc`. Initial estimation of the central pressure. To be updated during the calculation.
- `opa`. Type of opacity law. Possible values are:

- `opal`. OPAL opacities.
- `houdek`. Houdek's interpolation of OPAL opacities (smoother), see Houdek and Rogl (1996), "On the accuracy of opacity interpolation schemes", *Bull. Ast. Soc. India*, **24**, 317.
- `kramer`. Kramer's opacity.

• `eos`. Type of equation of state. Possible values are:

- `opal`. OPAL equation of state.
- `ideal`. Ideal gas.
- `ideal+rad`. Ideal gas with radiation.

• `nuc`. Type of nuclear reactions. Possible values are:

- `simple`. Simplified formulation of pp and CNO cycles.
- `cesam`. NACRE reaction rates as implemented in the ppcno9 chain of the CESAM code.

• `atm`. Type of atmosphere. At the moment, only `simple` is implemented.

• `core_convec` (default 1). Use 0 to disable core convection.

• `min_core_size` (default 0.01). The minimum size of the convective core in fraction of the polar radius).

If some parameters are omitted, the program will take the value from the input file (set with `-input_file` or `-i`) or from the default parameters file in `ester/config/1d_default.par` when no input file is specified.

2.4. `ester 2d` INPUT PARAMETERS

Note that the input of `ester 2d` can be a non-rotating 1D model calculated with `ester 1d`.

The program `ester 2d` admits the same parameters than `ester 1d` plus some extra specific options:

- `nth`. The number of grid points in latitude.
- `nex`. Number of radial points in the external domain.
- `Omega_bk`. Angular velocity at the equator in units of the critical velocity $\Omega_c = \sqrt{\frac{GM}{R_e^3}}$.
- `Ekman`. Ekman number.

2.5. Some recipes

The typical workflow to calculate a model starts with the calculation of the corresponding 1D model and using it as an input for `star2d`. For example, to calculate the structure of a $5M_\odot$ star with OPAL opacity rotating at with $\Omega = 0.5\Omega_c$ we can do:

```
$ ester 1d -M 5 -opa opal -o model1d
$ ester 2d -i model1d -nth 24 -Omega_bk 0.5 -o model2d
```

As the code uses the Newton's method, sometimes it is not possible to converge to a solution if the initial estimation is too far from it. In this case we can use some intermediate steps. For example, if we want to calculate the structure of a $2.5M_\odot$ star rotating with $\Omega = 0.9\Omega_c$, we should probably do

`$ ester 1d -M 2.5 -o model1d`	(Start with a non rotating 1D model)
`$ ester 2d -i model1d -nth 24 -Omega_bk 0.5 -o model2d`	(Using an intermediate value for rotation)
`$ ester 2d -i model2d -nth 32 -Omega_bk 0.9 -o model2d`	(Calculating the final model)

Executing `ester 2d` with `maxit=0` can be used to interpolate a model without recalculating it.

`$ ester 2d -i model -npts` *npts_new* `-nth` *nth_new* `-o model_interp -maxit 0`

Pressing Ctrl-C at any time during the execution of `ester 2d` will terminate the program, giving the possibility of finishing the current iteration and write the result in the output file.

2.6. Spatial resolution and memory requirements

The ESTER code uses a direct method to solve the equations of structure of a star. This type of method involves the factorization of a big matrix that arises from the discretization of the equations. The main drawback is that memory requirements are high, but the stiffness of the equations prevents the convergence of an iterative (matrix-free) method that will be more memory efficient.

The memory needed by the calculation can be estimated as:

$$\text{RAM Used} \gtrsim 25 \times n_d \times n_r^2 \times n_\theta^2 \times 8\,\text{bytes}$$

where

n_d: Number of domains
n_r: Number of radial points per domain
n_θ: Number of points in latitude

Of course, this is a lower limit, the actual memory used can be between ~10% and ~50% higher than this value. This overhead increases with the number of domains and decreases with the overall number of points. As an example, the following table shows the memory usage for some configurations (this is only approximated, real values are machine-dependent):

n_d	n_r	n_θ	RAM (estimated)	RAM (real)	Overhead
8	30	24	791 Mb	986 Mb	25%
8	50	32	3.81 Gb	4.27 Gb	12%
32	25	32	3.81 Gb	4.61 Gb	21%

The first case correspond to the default values and, in most cases, gives a decent representation of the structure of the star for moderate rotation rates. If more precision is required, we need to increase the resolution. In particular, to achieve a good precision for high rotation values, the number of points in latitude should be incremented.

As seen in the second and third cases in the table, the total number of radial points $n_d \times n_r$ can be increased, without affecting considerably the memory usage, just by increasing the number of domains. Or equivalently, we can reduce the memory requirements by distributing the radial points over more domains.

As a rule of thumb, if the number of domains is doubled, keeping the total number of radial points, the memory required is reduced in a factor $\sqrt{2}$. However this does not necessarily imply an improvement in precision, as the order of the integration method is also reduced, so it should be used carefully.

2.6.1. Estimating the precision of the output model

The ESTER code uses spectral methods to calculate the structure of the star. The star is subdivided in a certain number of domains and, in each domain, the variables are represented by a double truncated series of orthogonal polynomials in the radial (ζ) and θ directions, for example

$$\rho(\zeta,\theta) = \sum_{i=0}^{n_r-1} \sum_{j=0}^{n_\theta-1} \rho_{ij} T_i(\zeta) P_j(\theta)$$

When n_r and n_θ are high enough, the spectral representation will converge to the exact function. To check the convergence of the coefficients $\rho_i j$, the corresponding normalised spectra for the density (ρ) are shown in the graphics output of ESTER (see 2.1 and 2.2).

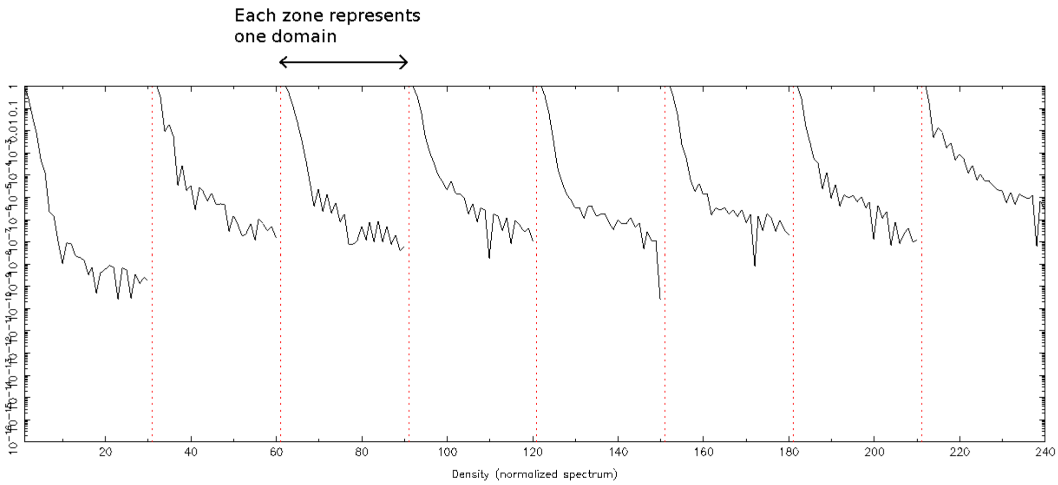


Figure 2.1: Example of spectrum plot showing the normalised coefficients ρ_i in a logarithmic scale in the graphical output of `ester 1d`.

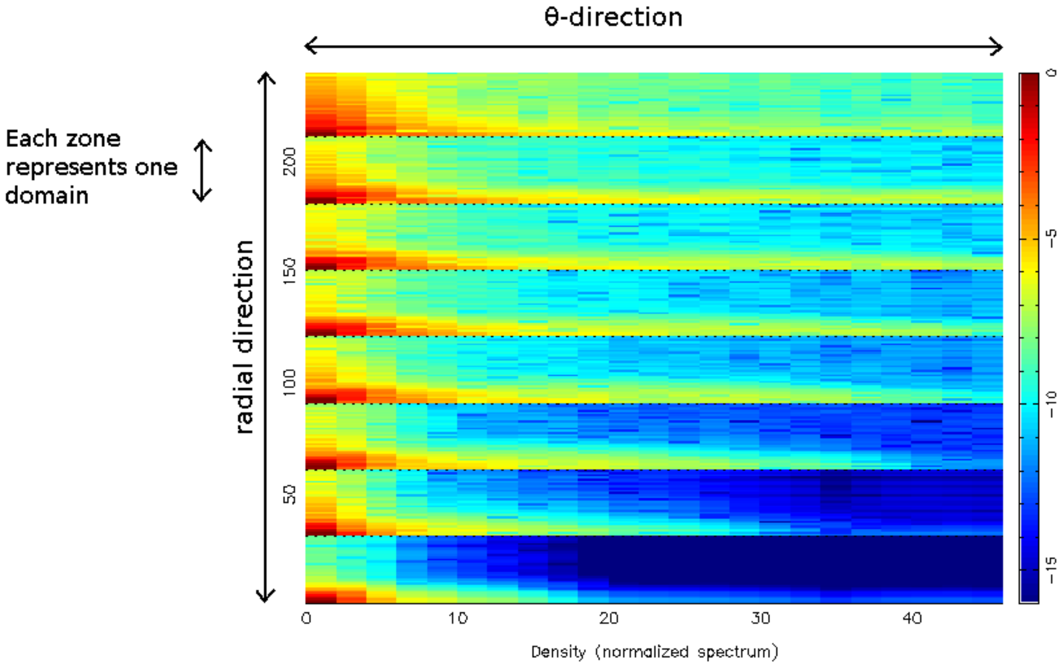


Figure 2.2: Example of 2D-spectrum showing the logarithm of the normalised coefficients ρ_{ij} in the graphical output of `ester 2d`.

There are two other indicators that can be used to estimate the quality of the solution:

- **The virial test**. It is the normalised residual resulting from the virial theorem. Ideally, it should be zero. It is mainly influenced by the internal layers and even with low resolution we can get very good values $\lesssim 10^{-9}$.
- **The energy test**. It is the relative difference between the luminosity of the star obtained as the integral over the volume of the energy generation rate and that obtained as the integral of the energy flux at the surface. It is highly influenced by the quality of the solution in the most external layers. Due to the use of tabulated opacities which are not quite smooth, the energy test will be always higher than 10^{-5}–10^{-6}.

In the following table, we can see an example of the precision achieved for three different configurations. The three configurations use approximately 1 Gb of RAM, but have different number of domains. We have used a model with 1600 radial points distributed over 16 domains and $n_\theta = 48$ (approx. 70 Gb of RAM) as a reference to calculate the errors.

n_d	n_r	n_θ	Virial test	Energy test	$\Delta\rho/\rho$	$\Delta R/R$	$\Delta T_{\rm eff}/T_{\rm eff}$	$\Delta\Omega/\Omega$
8	30	24	1.848e-09	1.192e-03	1.613e-04	5.157e-05	3.338e-04	3.739e-04
16	21	24	1.851e-11	1.726e-04	1.065e-04	4.702e-05	1.215e-04	4.446e-04
32	15	24	3.377e-10	3.447e-05	2.780e-05	2.021e-06	1.736e-05	3.095e-04

2.7. Generating custom output files

The output files generated by `ester 1d` and `ester 2d` contain just the minimal information necessary to reconstruct the model. However, sometimes a more detailed output is required. This can be done using `ester output`. This program reads a template from the standard input and write the result in the standard output. A typical call would be

`$ ester output` *model_file* `<` *template_file* `>` *output_file*

The template file is a regular text file with the following rules:

- Plain text is copied from the template to the output file. It cannot contain the reserved characters `$` and \.
- Line breaks are ignored. To insert a line break in the output file you have to insert a blank line in the template.
- Variables from the model are written in the form `${`*var*`,`*fmt*`}`, where *var* is the code for the variable (see table below) and *fmt* is a valid format for the C function *printf* (e.g. `%d` for an integer, `%f` for float, `%e` for exponential notation). If `fmt` is omitted `${`*var*`}` the variable is written in binary format.

Table 2.1: Non-exhaustive list of variables codes for the model in the template file. Dimensional quantities are in cgs

Code	**Description**	**star1d**	**star2d**
nr	# of radial points	*	*
nth	# of points in latitude		*
ndomains	# of domains	*	*
npts	# of radial points in each domain	*	*
xif	Position of each domain	*	*
nex	# of radial points in the external domain		*
surff	Parameter surff (see above)	*	*
conv	# of convective domains	*	*
Omega	Angular velocity at the equator		*
Omega_bk	Angular velocity at the equator in units of the critical velocity		*
Omegac	Critical velocity $\Omega_c = \sqrt{\frac{GM}{R_e^3}}$		*
X	Hydrogen abundance	*	*
Z	Metal abundance	*	*
Xc	Fraction of X at the convective core	*	*
rhoc	Central density	*	*
Tc	Central temperature	*	*
pc	Central pressure	*	*
M	Mass	*	*
R	(Polar) Radius	*	*
Rp	Polar radius		*
Re	Equatorial radius		*
L	Luminosity	*	*

M/M_SUN	Mass in solar units	*	*
R/R_SUN	(Polar) Radius in solar units	*	*
Rp/R_SUN	Polar radius in solar units		*
Re/R_SUN	Equatorial radius in solar units		*
L/L_SUN	Luminosity in solar units	*	*
r	Radius	*	*
z	Radial variable	*	*
th	Colatitude		*
rex	External radius		*
phi	Gravitational potential	*	*
phiex	Gravitational potential of the external domain		*
rho	Density	*	*
p	Pressure	*	*
T	Temperature	*	*
w	Angular velocity		*
G	Stream function for the meridional circulation		*
Xr	Hydrogen abundance $X(r,\theta)$	*	*
N2	Squared Brunt-Väisälä frequency (in $\mathrm{rd}^2/\mathrm{s}^2$)	*	*
opa	Type of opacity	*	*
opa.k	Rosseland mean opacity	*	*
opa.xi	Thermal conductivity (χ)	*	*
opa.dlnxi_lnT	$\left(\frac{\partial \log \chi}{\partial \log T}\right)_{\rho,\mu}$	*	*
opa.dlnxi_lnrho	$\left(\frac{\partial \log \chi}{\partial \log \rho}\right)_{T,\mu}$	*	*
eos	Type of equation of state	*	*
eos.G1	Γ_1	*	*
eos.cp	c_p	*	*
eos.del_ad	∇_{ad}	*	*
eos.G3_1	$\Gamma_3 - 1$	*	*
eos.cv	c_v	*	*
eos.prad	Radiation pressure	*	*
eos.chi_T	$\chi_T = \left(\frac{\partial \log p}{\partial \log T}\right)_{\rho,\mu}$	*	*
eos.chi_rho	$\chi_\rho = \left(\frac{\partial \log p}{\partial \log \rho}\right)_{T,\mu}$	*	*
eos.d	$d = \frac{\chi_T}{\chi_\rho} = -\left(\frac{\partial \log \rho}{\partial \log T}\right)_{p,\mu}$	*	*
nuc.eps	Energy generation rate per unit mass	*	*
nuc.pp	Energy generation rate per unit mass (pp-chain)	*	*
nuc.cno	Energy generation rate per unit mass (CNO cycle)	*	*
Teff	Effective temperature at the surface $T_{\mathrm{eff}}(\theta)$	*	*
gsup	Effective gravity at the surface $g_{\mathrm{eff}}(\theta)$	*	*
D	Radial differentiation matrix $\frac{\partial}{\partial\zeta}$ for 2D models, $\frac{\mathrm{d}}{\mathrm{d}r}$ for 1D models	*	*
I	Radial integration matrix	*	*
Dex	Radial differentiation matrix for the external domain		*

Dt	Angular differentiation matrix $\frac{\partial}{\partial\theta}$ for symmetric variables		*
Dtodd	Angular differentiation matrix for antisymmetric variables		*
Dt2	Second order angular differentiation matrix for symmetric variables		*
It	Angular integration matrix over $\mu = \cos\theta$		*

For 2D variables, their values at the collocation points are written in the output file in matrix form. Each line corresponds to a different value of the colatitude θ (i.e. a different column), starting at the equator.

$$\begin{array}{cccc} p(\zeta_0,\theta_0) & p(\zeta_1,\theta_0) & p(\zeta_2,\theta_0) & \cdots \\ p(\zeta_0,\theta_1) & p(\zeta_1,\theta_1) & p(\zeta_2,\theta_1) & \cdots \\ p(\zeta_0,\theta_2) & p(\zeta_1,\theta_2) & p(\zeta_2,\theta_2) & \cdots \\ \vdots & \vdots & \vdots & \end{array}$$

Being ζ the radial spheroidal coordinate. Similarly, 1D variables can be seen as a column vector and are written in one line in the output file, terminated by a new line character. This behavior can be inverted by writing this line in the template file

```
\conf{transpose=1}
```

After this command, the variables will be written row wise, i.e. one line for each value of the radial coordinate. Note that it does not affect variables written in binary format, which are always column wise. To recover the original behaviour we use

```
\conf{transpose=0}
```

The original grid does not contain points in the equator and the pole. If we want the values at this points we should write

```
\conf{equator=1}
\conf{pole=1}
```

By default, the output uses cgs units. If we want the normalized values used internally by the code, we simply put

```
\conf{dim=0}
```

These control commands can be written anywhere in the template file, in separated lines, affecting only the code that appears below them.

Let's see an example.

Template file:

```
Model of ${M/M_SUN,%.2f} solar masses and R=${R,%e} cm

rotating with Omega=${Omega_bk,%f} Omegac

${nr,%d} radial points and
${nth,%d} latitudinal points

```

```
\conf{pole=1}
\conf{equator=1}
r:

${r,%e}
Pressure:

${p,%.14e}
```

Output file:

```
Model of 2.50 solar masses and R=1.219822e+11 cm
rotating with Omega=0.900000 Omegac
240 radial points and 32 latitudinal points
r:
0.000000e+00 4.944313e+07 1.971944e+08 4.415355e+08 7.796539e+08 ...
0.000000e+00 4.944313e+07 1.971944e+08 4.415355e+08 7.796539e+08 ...
0.000000e+00 4.944313e+07 1.971944e+08 4.415354e+08 7.796533e+08 ...
0.000000e+00 4.944313e+07 1.971944e+08 4.415352e+08 7.796523e+08 ...
[...]
Pressure:
1.61049808835808e+17 1.61048890365891e+17 1.61035199104197e+17 ...
1.61049808835808e+17 1.61048890354742e+17 1.61035198927083e+17 ...
1.61049808835808e+17 1.61048890265707e+17 1.61035197512689e+17 ...
1.61049808835808e+17 1.61048890088480e+17 1.61035194697311e+17 ...
[...]
```

2.8. Python module

A basic python module for reading the models is included in the distribution. It is located in `ester/python/star.py`. At the moment it only works for models calculated using `ester 2d`. The variables in the models are defined as *numpy* arrays. Here is a little example:

```python
import sys
sys.path.append( 'path_to/ester/python' ) # include the full path to the module

from star import * # Loads the module

A=star2d( 'model_file' )  # Loads a model

print A.p[0,0] # Prints the central pressure

A.draw(A.w) # Makes a plot of the differential rotation
show() # Needed in non-interactive mode of matplotlib
```

Note that "dotted variables" like `opa.k` are accessed via `A.opa_k` under python.

3 General structure of the code

The code is divided in several libraries. Each library implements one ore more classes designed to handle one particular aspect of the calculation.

- **matrix**. Matrix algebra.
- **numdiff**. Implements Gauss-Legendre and multi-domain Gauss-Lobatto numerical differentiation.
- **mapping**. Defines the mapping in spheroidal coordinates $r(\zeta,\theta)$.
- **solver**. Resolution of systems of linear differential equations in 2D.
- **physics**. Calculation of physical quantities (opacity, equation of state, nuclear reaction rates).
- **star**. Provides objects and functions to calculate the structure of a star in 1D and 2D.
- **global**. Definition of global variables, e.g. physical and mathematical constants.
- **graphics**. Provides graphical output through `pgplot`.
- **parser**. Parsing of configuration files and command-line arguments and file input/output.

4

Matrix Algebra. The `matrix` library.

To facilitate the work with matrices in C++, the `matrix` library provides two classes:

- `matrix` for regular matrices
- `matrix_block_diag` for block diagonal matrices

The function prototypes are defined in the header file `matrix.h`.

4.1. Matrix creation and manipulation

Regular matrices are defined as objects of the `matrix` class. For example, the sentence:

```
matrix a(3,4);
```

creates a matrix `a` with 3 rows and 4 columns. If the size is not specified,

```
matrix a;
```

a 1x1 matrix is created. The size of the matrix can be modified using the method `dim`

```
a.dim(3,4);
```

or, if the total number of elements does not change, using `redim`

```
a.redim(1,12);
```

With `redim` the element values are also preserved. The number of rows and columns of a matrix object can be retrieved using the methods `nrows()` and `ncols()`. For example

```
int n,m;
matrix a(3,4);

n=a.nrows();
m=a.ncols();
```

in this example $n = 3$ and $m = 4$.

The elements of the matrix can be indexed using parenthesis. Note that, as regular C arrays, the index of the first element is 0. There are also methods for extracting parts of the matrix. Let's see an example

```
matrix a(3,3),row,col,block;
double elem;

a(0,0)=1;a(0,1)=2;a(0,2)=3;
a(1,0)=4;a(1,1)=5;a(1,2)=6;
a(2,0)=7;a(2,1)=8;a(2,2)=9;

elem=a(1,2); // elem=6
row=a.row(1); // Extracts the second row
col=a.col(0); // Extracts the first column
block=a.block(0,1,1,2); // Extracts the block (0-1,1-2)
```

After running the example, the contents of the different matrices will be

$$\mathtt{a} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

$$\mathtt{row} = \begin{pmatrix} 4 & 5 & 6 \end{pmatrix} \qquad \mathtt{col} = \begin{pmatrix} 1 \\ 4 \\ 7 \end{pmatrix} \qquad \mathtt{block} = \begin{pmatrix} 2 & 3 \\ 5 & 6 \end{pmatrix}$$

We can also insert parts of the matrix using the methods `setrow`, `setcol` and `setblock`.

```
matrix a(3,3),b;

b=ones(1,3); // Creates a 1x3 array of all ones
a.setrow(0,b);

b=ones(3,1);
a.setcol(2,b);

b=ones(2,3);
a.setblock(1,2,0,2,b);
```

Negative indices are interpreted starting from the end of the matrix. For example `a.row(-1)` returns the last row of the matrix `a`.

Indexing with only one parameter is also possible, being `a(i,j)` equivalent to `a(j*a.nrows()+i)`. This makes sense when working with row or column vectors, if we define

```
matrix row(1,5),col(5,1);
```

then `row(i)` is equivalent to `row(1,i)` and `col(i)` is equivalent to `col(i,1)`.

4.2. File input/output

The method `write` writes a matrix in a file, the syntax is

```
write(FILE *fp, char mode)
```

Here, `mode` can be `'t'` for text output or `'b'` for binary output. Default is `'t'`. The matrix is written in column-wise order, i.e. each line represents a column of the matrix. When called without arguments `write()`, it writes the matrix in the standard output.

To read a matrix from a file we use the method `read`.

```
read(int nrow, int ncol, FILE *fp, char mode)
```

Where we must specify the size of the matrix.

In the following example, we will write a matrix to a file and read it again.

```
#include<stdio.h>
#include"matrix.h"
int main() {
    FILE *fp;
    matrix a(2,3);

    a(0,0)=1;a(0,1)=2;a(0,2)=3;
    a(1,0)=4;a(1,1)=5;a(1,2)=6;

    // Write the matrix to a file in binary mode
    fp=fopen("matrix.dat","wb");
    a.write(fp,'b');
    fclose(fp);

    // Read the matrix from file
    fp=fopen("matrix.dat","rb");
    a.read(2,3,fp,'b');
    fclose(fp);

    return 0;
}
```

We can write a matrix on the screen in a more convenient format using `write_fmt`. For the previous example the sentence

```
a.write_fmt("%.2f");
```

will produce the following output

```
1.00 2.00 3.00
4.00 5.00 6.00
```

4.3. Operators

Element-wise operators for the matrix class:

<table>
<tr><td><code>a=b</code></td><td>Assignment</td></tr>
<tr><td><code>a+b</code></td><td>Addition</td></tr>
<tr><td><code>a-b</code></td><td>Subtraction</td></tr>
<tr><td><code>a*b</code></td><td>Element-wise multiplication</td></tr>
<tr><td><code>a/b</code></td><td>Element-wise division</td></tr>
<tr><td><code>a+=b</code></td><td>Equivalent to <code>a=a+b</code></td></tr>
<tr><td><code>a-=b</code></td><td>Equivalent to <code>a=a-b</code></td></tr>
<tr><td><code>a*=b</code></td><td>Equivalent to <code>a=a*b</code></td></tr>
<tr><td><code>a/=b</code></td><td>Equivalent to <code>a=a/b</code></td></tr>
<tr><td><code>+a</code></td><td>Unary plus</td></tr>
<tr><td><code>-a</code></td><td>Unary minus</td></tr>
<tr><td><code>a==b</code></td><td>Comparison: Equal to</td></tr>
<tr><td><code>a!=b</code></td><td>Comparison: Not equal to</td></tr>
<tr><td><code>a>b</code></td><td>Comparison: Greater than</td></tr>
<tr><td><code>a<b</code></td><td>Comparison: Less than</td></tr>
<tr><td><code>a>=b</code></td><td>Comparison: Greater than or equal to</td></tr>
<tr><td><code>a<=b</code></td><td>Comparison: Less than or equal to</td></tr>
<tr><td><code>a&&b</code></td><td>Logical AND</td></tr>
<tr><td><code>a||b</code></td><td>Logical OR</td></tr>
</table>

The operands `a` and `b` can be either matrices or scalars. Element-wise operators are performed element by element. For example if we define

```
c=a*b;
```

the elements of the new matrix `c` will be

```
c(i,j)=a(i,j)*b(i,j)
```

obviously, the two matrices must have the same size. There is one exception, when one or both of the dimensions are one, for example if `a` is (n,m) and `b` is (1,m) the matrix `c` will be (n,m) with elements

```
c(i,j)=a(i,j)*b(j)
```

also if `a` is (n,1) and `b` is (1,m), `c` will be (n,m) with

```
c(i,j)=a(i)*b(j)
```

The comparison operator `==` compares two matrices element by element, so the result is a new matrix whose elements are 1 if the corresponding elements of `a` and `b` are equal or 0 if they are different. If we want to know if two matrices are completely equal, we can use the function `isequal(a,b)` that returns 1 if `a` and `b` are the same and 0 otherwise.

Matrix product are indicated with a comma ",". The product of matrices `a` and `b` are expressed as:

```
c=(a,b);
```

The operation should be put in parentheses when necessary to avoid ambiguity. Note that the operator "," in C has the lowest precedence, for example

```
(2*a,b+c,d)
```

is equivalent to

```
( (2*a) , ( (b+c) ,d ) )
```

4.4. Block diagonal matrices

Another type of object included in the library are the block diagonal matrices. An object of this class has the following structure

$$M = \begin{pmatrix} M_0 & 0 & \cdots & 0 \\ 0 & M_1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & M_{n-1} \end{pmatrix}$$

where the M_i are also matrices. Although the definition of a block diagonal matrix requires the blocks M_i to be square, in the current implementation they are allowed to have any size.

A block diagonal matrix is defined using the sentence

```
matrix_block_diag D;
```

An optional argument can be included to specify the number of blocks in the matrix (default is 1)

```
matrix_block_diag D(4);
```

Alternatively, the number of blocks can be changed using the sentence

```
D.set_nblocks(4);
```

In order to access the different blocks, we use the method `block(int i)`, for exmaple

```
matrix_block_diag D(3);
matrix a,b;

a=ones(2,2);
D.block(0)=a;
b=D.block(0);
```

Individual elements can also be indexed using parentheses `D(i,j)`, as with regular matrices.

A number of operators are defined in the `matrix_block_diag` class:

Operator	Operands type	Return type	Description
a=b	`matrix_block_diag`	`matrix_block_diag`	Assignment
a+b	`matrix_block_diag`	`matrix_block_diag`	Addition
a-b	`matrix_block_diag`	`matrix_block_diag`	Subtraction
+a	`matrix_block_diag`	`matrix_block_diag`	Unary plus
-a	`matrix_block_diag`	`matrix_block_diag`	Unary minus
a*b	`matrix_block_diag`	`matrix_block_diag`	Element-wise multiplication
	`matrix_block_diag` & `matrix`	`matrix_block_diag`	
	`matrix` & `matrix_block_diag`	`matrix_block_diag`	
	`matrix_block_diag` & `double`	`matrix_block_diag`	
	`double` & `matrix_block_diag`	`matrix_block_diag`	
a/b	`matrix_block_diag` & `matrix`	`matrix_block_diag`	Element-wise division
	`matrix_block_diag` & `double`	`matrix_block_diag`	
(a,b)	`matrix_block_diag`	`matrix_block_diag`	Matrix product
	`matrix_block_diag` & `matrix`	`matrix_block_diag`	
	`matrix` & `matrix_block_diag`	`matrix_block_diag`	

For element-wise operators between `matrix_block_diag` objects, both objects must have exactly the same structure. Matrix product is also performed block by block, so the structure of the operands must be compatible.

A `matrix_block_diag` object can be converted in a `matrix` object using type casting.

```
matrix a;
matrix_block_diag D;

a=(matrix) D;
```

4.5. Reference

Matrix manipulation

dim(n,m)
redim(n,m)
nrows()
ncols()
row(n)
col(n)
block(n1,n2,m1,m2)
block_step(n1,n2,nstep,m1,m2,mstep)
setrow(n,A)
setcol(n,A)
setblock(n1,n2,m1,m2,A)
setblock_step(n1,n2,nstep,m1,m2,mstep,A)
transpose()
fliplr()
flipud()
data()
swap()
zero(n,m)

File input/output

write(fp,mode)
read(n,m,fp,mode)
write_fmt(format,fp)

Special matrices

eye(n)
zeros(n,m)
ones(n,m)
random_matrix(n,m)
vector(x0,x1,n)
vector_t(x0,x1,n)

Matrix functions

max(A)
min(A)
sum(A)
mean(A)
max(A,B)
min(A,B)
exist(A)
isequal(A,B)
solve(b)
inv()

Mathematical functions

cos(x)
sin(x)
tan(x)
acos(x)
asin(x)
atan(x)
atan2(y,x)
cosh(x)
sinh(x)
tanh(x)
exp(x)
log(x)
log10(x)
sqrt(x)
abs(x)
pow(x,y)
round(x)
floor(x)
ceil(x)

Block diagonal matrices

set_nblocks(n)
block(n)
nblocks()
nrows()
ncols()
row(n)
transpose()
eye(D)

4.5.1. A note about methods and functions

The subroutines are divided in two types: functions and methods. Contrary to functions, methods belong to the object and they are called using a different syntax. For example if `met` is a method of the object `a` that takes one argument `b` and returns a value `c`, we use the sentence

```
c=a.met(b)
```

The same subroutine implemented as a function will be

```
c=met(a,b)
```

When using pointers, the dot is replaced by `->`, then if `p=&a` the sentence above is equivalent to

```
c=p->met(b)
```

The parenthesis are needed even if the method takes no arguments, i.e. `a.method_without_args()`.

4.5.2. Matrix manipulation

---

dim(n,m)

Type:	Method
Inputs:	n (int): Number of rows m (int): Number of columns
Output:	Reference to current object
Description:	Changes the dimensions of the matrix object.

---

redim(n,m)

Type:	Method
Inputs:	n (int): Number of rows m (int): Number of columns
Output:	Reference to current object
Description:	Changes the dimensions of the matrix object. The total number elements must not change. Element values are preserved.

---

nrows()

Type:	Method
Inputs:	None
Output:	int
Description:	Returns the number of rows of the matrix.

---

ncols()

Type:	Method
Inputs:	None
Output:	int
Description:	Returns the number of columns of the matrix.

---

row(n)

Type:	Method
Inputs:	n (int): Row index
Output:	matrix
Description:	Extracts row n from matrix.

col(n)

Type:	Method
Inputs:	n (int): Column index
Output:	matrix
Description:	Extracts column n from matrix.

block(n1,n2,m1,m2)

Type:	Method
Inputs:	n1 (int): First row index n2 (int): Last row index m1 (int): First column index m2 (int): Last column index
Output:	matrix
Description:	Extracts the block contained between the rows n1 and n2 and the columns m1 and m2.

block_step(n1,n2,nstep,m1,m2,mstep)

Type:	Method
Inputs:	n1 (int): First row index n2 (int): Last row index nstep (int): Row increment m1 (int): First column index m2 (int): Last column index mstep (int): Column increment
Output:	matrix
Description:	Extracts the block contained between the rows n1 and n2 and the columns m1 and m2 using increments nstep and mstep.

setrow(n,A)

Type:	Method
Inputs:	n (int): Row index A (matrix)
Output:	Reference to current object
Description:	Inserts matrix A at row n.

setcol(n,A)

Type:	Method
Inputs:	n (int): Column index A (matrix)
Output:	Reference to current object
Description:	Inserts matrix A at column n.

setblock(n1,n2,m1,m2,A)

Type:	Method
Inputs:	n1 (int): First row index n2 (int): Last row index m1 (int): First column index m2 (int): Last column index A (matrix)
Output:	Reference to current object
Description:	Inserts matrix A between the rows n1 and n2 and the columns m1 and m2.

setblock_step(n1,n2,nstep,m1,m2,mstep,A)

Type:	Method
Inputs:	n1 (int): First row index n2 (int): Last row index nstep (int): Row increment m1 (int): First column index m2 (int): Last column index mstep (int): Column increment A (matrix)
Output:	Reference to current object
Description:	Inserts matrix A between between the rows n1 and n2 and the columns m1 and m2 using increments nstep and mstep.

transpose()

Type:	Method
Inputs:	None
Output:	matrix
Description:	Returns the tranpose of the object. Does not modify the original matrix.

fliplr()

Type:	Method
Inputs:	None
Output:	matrix
Description:	Flip columns in the left-right direction. Does not modify the original matrix.

flipud()

Type:	Method
Inputs:	None
Output:	matrix
Description:	Flip rows in the up-down direction. Does not modify the original matrix.

data()

Type:	Method
Inputs:	None
Output:	Pointer to double
Description:	Returns a pointer to the first element in the matrix. The elements are stored consecutively in column order.

swap()

Type:	Method
Inputs:	matrix
Output:	None
Description:	Swaps the contents of the current matrix object and the one used as argument.

zero(n,m)

Type:	Method
Inputs:	n (int): Number of rows m (int): Number of columns
Output:	None
Description:	Creates a nxm matrix of all zeros. Note that a.zero(n,m) is equivalent to a=zeros(n,m) but avoids the creation of an intermediate object, saving memory for large matrices.

4.5.3. File input/output

write(fp,mode)

Type:	Method
Inputs:	fp (FILE *): File pointer (optional, default=stdout) mode (char): Write mode (optional, default='t')
Output:	int
Description:	Writes a matrix in the file pointed by fp in text mode (mode='t') or binary mode (mode='b'). The matrix is written in column order. Returns 1 on success, 0 otherwise.

read(n,m,fp,mode)

Type: Method

Inputs: n (int): Number of rows
m (int): Number of columns
fp (FILE *): File pointer
mode (char): Write mode (optional, default='t')

Output: int

Description: Reads a nxm matrix from the file pointed by fp in text mode (mode='t') or binary mode (mode='b'). The matrix is read in column order. Returns 1 on success, 0 otherwise.

---

write_fmt(format,fp)

Type: Method

Inputs: format (char *): Format string
fp (FILE *): File pointer (optional, default=stdout)

Output: None

Description: Writes a matrix in the file pointed by fp using given format. The matrix is ordered such that each line represents a row.

4.5.4. Special matrices

---

eye(n)

Type: Function

Inputs: n (int): Number of rows

Output: matrix

Description: Returns the nxn identity matrix.

---

zeros(n,m)

Type: Function

Inputs: n (int): Number of rows
m (int): Number of columns

Output: matrix

Description: Returns a nxm matrix of all zeros.

---

ones(n,m)

Type: Function

Inputs: n (int): Number of rows
m (int): Number of columns

Output: matrix

Description: Returns a nxm matrix of all ones.

random_matrix(n,m)

Type:	Function
Inputs:	n (int): Number of rows m (int): Number of columns
Output:	matrix
Description:	Returns a nxm matrix with random values between 0 and 1.

vector(x0,x1,n)

Type:	Function
Inputs:	x0 (double): Minimum value x1 (double): Maximum value n (int): Number of elements
Output:	matrix
Description:	Returns a row vector with n equally spaced elements between x0 and x1.

vector_t(x0,x1,n)

Type:	Function
Inputs:	x0 (double): Minimum value x1 (double): Maximum value n (int): Number of elements
Output:	matrix
Description:	Returns a column vector with n equally spaced elements between x0 and x1.

4.5.5. Matrix functions

max(A)

Type:	Function
Inputs:	A (matrix)
Output:	double
Description:	Returns the maximum value.

min(A)

Type:	Function
Inputs:	A (matrix)
Output:	double
Description:	Returns the minimum value.

sum(A)

Type:	Function
Inputs:	A (matrix)
Output:	double
Description:	Returns the sum of the matrix elements.

mean(A)

Type:	Function
Inputs:	A (matrix)
Output:	double
Description:	Returns the mean value of the matrix elements.

max(A,B)

Type:	Function
Inputs:	B (matrix) B (matrix)
Output:	matrix
Description:	Compares the matrices a and b and returns a new matrix C containing the larger values of each pair of elements C(i,j)=max(A(i,j),B(i,j)).

min(A,B)

Type:	Function
Inputs:	A (matrix) B (matrix)
Output:	matrix
Description:	Compares the matrices A and B and returns a new matrix C containing the smaller values of each pair of elements C(i,j)=min(A(i,j),B(i,j)).

exist(A)

Type:	Function
Inputs:	A (matrix)
Output:	int
Description:	Returns 1 if any of the elements of A is not zero, 0 otherwise. It is often used in constructions of type `if (exist(condition))...`, where `condition` is a valid comparison. For example `if (exist(A<0))...`.

isequal(A,B)

Type:	Function
Inputs:	A (matrix) B(matrix)
Output:	int
Description:	Returns 1 if matrices A and B contain exactly the same values, 0 otherwise.

solve(b)

Type:	Method
Inputs:	b (matrix)
Output:	matrix
Description:	`x=A.solve(b)` solves the linear system $Ax = b$ and returns matrix x.

inv()

Type:	Method
Inputs:	None
Output:	matrix
Description:	Returns the inverse of the current matrix object. The original matrix is not modified.

4.5.6. Mathematical functions

cos(x)

Type:	Function
Inputs:	x (matrix)
Output:	matrix
Description:	Returns the cosine of x. x must be expressed in radians.

sin(x)

Type:	Function
Inputs:	x (matrix)
Output:	matrix
Description:	Returns the sine of x. x must be expressed in radians.

tan(x)

Type:	Function
Inputs:	x (matrix)
Output:	matrix
Description:	Returns the tangent of x. x must be expressed in radians.

acos(x)

Type:	Function
Inputs:	x (matrix)
Output:	matrix
Description:	Returns the arc cosine of x in radians.

asin(x)

Type:	Function
Inputs:	x (matrix)
Output:	matrix
Description:	Returns the arc sine of x in radians.

atan(x)

Type:	Function
Inputs:	x (matrix)
Output:	matrix
Description:	Returns the arc tangent of x in radians.

atan2(y,x)

Type:	Function
Inputs:	y (matrix or double) x (matrix or double)
Output:	matrix
Description:	Returns the arc tangent of y/x in radians. Uses the sign of y and x to determine the quadrant.

cosh(x)

Type:	Function
Inputs:	x (matrix)
Output:	matrix
Description:	Returns the hyperbolic cosine of x. x must be expressed in radians.

sinh(x)

Type:	Function
Inputs:	x (matrix)
Output:	matrix
Description:	Returns the hyperbolic sine of x. x must be expressed in radians.

tanh(x)

Type:	Function
Inputs:	x (matrix)
Output:	matrix
Description:	Returns the hyperbolic tangent of x. x must be expressed in radians.

exp(x)

Type:	Function
Inputs:	x (matrix)
Output:	matrix
Description:	Returns e^x.

log(x)

Type:	Function
Inputs:	x (matrix)
Output:	matrix
Description:	Returns $\log(x)$.

log10(x)

Type:	Function
Inputs:	x (matrix)
Output:	matrix
Description:	Returns $\log_{10}(x)$.

sqrt(x)

Type:	Function
Inputs:	x (matrix)
Output:	matrix
Description:	Returns $\sqrt{x}$.

abs(x)

Type:	Function
Inputs:	x (matrix)
Output:	matrix
Description:	Returns the absolute value of x.

pow(x,y)

Type:	Function
Inputs:	x (matrix or double) y (matrix or double)
Output:	matrix
Description:	Returns x^y.

round(x)

Type:	Function
Inputs:	x (matrix)
Output:	matrix
Description:	Rounds the elements of x to the nearest integer.

floor(x)

Type:	Function
Inputs:	x (matrix)
Output:	matrix
Description:	Rounds the elements of x to the nearest integer below the current value.

ceil(x)

Type:	Function
Inputs:	x (matrix)
Output:	matrix
Description:	Rounds the elements of x to the nearest integer above the current value.

4.5.7. Block diagonal matrices

set_nblocks(n)

Type:	Method
Inputs:	n (int): Number of blocks
Output:	Reference to current object
Description:	Changes the number of blocks of the matrix_block_diag object.

block(n)

Type:	Method
Inputs:	n (int): Block number
Output:	Reference to matrix
Description:	Returns a reference to the matrix in the block number n.

nblocks()

Type:	Method
Inputs:	None
Output:	int
Description:	Returns the number of blocks.

nrows()

Type:	Method
Inputs:	None
Output:	int
Description:	Returns the total number of rows.

ncols()

Type:	Method
Inputs:	None
Output:	int
Description:	Returns the total number of columns.

row(n)

Type:	Method
Inputs:	n (int): Row number
Output:	int
Description:	Extracts the row n.

transpose()

Type:	Method
Inputs:	None
Output:	matrix_block_diag
Description:	Calculates the transpose.

eye(D)

Type:	Function
Inputs:	D (matrix_block_diag)
Output:	matrix_block_diag
Description:	Returns the identity block matrix with the same structure as D.

5 Numerical differentiation

5.1. Introduction

Numerical differentiation refers to the algorithms for estimating the derivative of a function using only its values at certain evaluation points.

The simplest method for evaluating the derivative is the finite difference method. Given a function f sampled at points x_i ($i = 0, \ldots, n-1$), its derivative is estimated using the formula

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}$$

For equally spaced points $x_{i+1} - x_i = h$ and the formula becomes

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h}$$

There are some variations of the finite difference formula, as for example the central difference

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2h}$$

When solving differential equations, it is a good idea to write this expression in matrix form:

$$\begin{pmatrix} f'(x_0) \\ f'(x_1) \\ f'(x_2) \\ f'(x_3) \\ \vdots \\ f'(x_{n-3}) \\ f'(x_{n-2}) \\ f'(x_{n-1}) \end{pmatrix} = \frac{1}{2h} \begin{pmatrix} -2 & 2 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & \cdots & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & \cdots & 0 & -2 & 2 \end{pmatrix} \begin{pmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ f(x_3) \\ \vdots \\ f(x_{n-3}) \\ f(x_{n-2}) \\ f(x_{n-1}) \end{pmatrix}$$

or, in more compact form

$$f'(x_i) = \sum_{j=0}^{n-1} D_{ij} f(x_j)$$

where D_{ij} is the differentiation matrix.

The finite difference method has order 2, which means that the error in the estimation of the derivative is proportional to h^2. In fact, it is possible to construct higher order methods using more points to estimate the derivative. As a rule of thumb, the order of the method is at least equal to the number of points used in the estimation of the derivative.

Unfortunately, high order methods using equally spaced points are affected by the Runge's phenomenon. Indeed, finite difference formulas of order n are obtained by interpolating the function between the points of interest using a polynomial of degree $n-1$. One of the main problems of polynomial interpolation using polynomials of high degree is the apparition of oscillations near the edges of the interval between the interpolation points. The amplitude of the oscillations increase with the degree of the polynomial and quickly degrades the derivative estimation for high order methods. This effect is known as the Runge's phenomenon.

Collocation or pseudospectral methods attempt to suppress the Runge's phenomenon by choosing a set of non-equally spaced points called collocation points.

5.1.1. Collocation/Pseudospectral methods

Collocation methods are high order methods for estimating the derivative of a function knowing its values at certain points called collocation points. The position of this points is different for each collocation method and is designed to suppress the Runge's phenomenon. A pseudospectral method with n points has order $2n$.

Each particular collocation method is associated with a family of orthogonal functions $P_l(x)$. This functions form a basis so, any arbitrary function $\phi(x)$ can be expressed as a linear combination of the basis functions

$$\phi(x) = \sum_{l=0}^{\infty} a_l P_l(x)$$

In practice, we will work with a finite discretization using n points, so the expansion is truncated to use only n basis functions, the fuction $\phi(x)$ is then approximated by

$$\phi^{(n)}(x) = \sum_{l=0}^{n-1} \phi_l P_l(x)$$

For regular functions and a well-adapted choice of the basis functions, collocation methods have exponential convergence, which means that the error in the approximation decreases exponentially with the the number of basis functions n.

The functions $P_l(x)$ are orthogonal against some scalar product $\langle P_l, P_m \rangle = \delta_{lm}$, then the coefficients on the expansion of $\phi(x)$ can be calculated as

$$\phi_l = \langle \phi(x), P_l(x) \rangle$$

For each family of basis functions it exists a formula of gaussian quadrature for calculating this scalar product, then

$$\phi_l = \sum_{j=0}^{n-1} w_j P_l(x_j) \phi(x_j)$$

where x_j and w_j are the nodes and weights of the corresponding gaussian quadrature. Note that x_j are the collocation points. The estimation of the first derivative at the collocation points can be obtained as

$$\begin{aligned}
\phi'(x_i) &= \sum_{l=0}^{n-1} \phi_l P_l'(x_i) \\
&= \sum_{l=0}^{n-1} \left(\sum_{j=0}^{n-1} w_j P_l(x_j) \phi(x_j) \right) P_l'(x_i) \\
&= \sum_{j=0}^{n-1} \left(\sum_{l=0}^{n-1} w_j P_l(x_j) P_l'(x_i) \right) \phi(x_j) \\
&= \sum_{j=0}^{n-1} D_{ij} \phi(x_j)
\end{aligned}$$

where $D_{ij} = \sum_{l=0}^{n-1} w_j P_l(x_j) P_l'(x_i)$ is the differentiation matrix. We see that the derivative of a discretized function can be calculated by doing a matrix product.

$$\Phi' = D\Phi$$

Similarly, the second derivative will be

$$\Phi'' = DD\Phi$$

5.1.2. Relation with spectral methods

Collocation methods are intimately related with spectral methods and share most of their properties. The main difference is that in spectral methods we work with the coefficients ϕ_l in the expansion of the functions contrarily to collocation methods that deal directly with the values of the function at the collocation points. This has a clear advantage when solving differential equations with variable coefficients. For example, the equation

$$\phi'(x) + a(x)\phi(x) = b(x)$$

will be discretized using spectral methods as

$$\sum_m \langle P_l, P_m' \rangle \phi_m + \sum_{m,k} \langle P_l, P_m P_k \rangle a_m \phi_k = b_l$$

While the first product $\langle P_l, P_m' \rangle$ use to be easy to calculate, this is not the case for the second one $\langle P_l, P_m P_k \rangle$. By contrast, for collocation methods the discretization is just

$$\sum_j D_{ij} \phi(x_j) + a(x_i)\phi(x_i) = b(x_i)$$

It is possible to pass from one representation to the other using projection matrices. To get the spectral coefficients of a function ϕ, we multiply by the projection matrix $\mathcal{P}_{ij}$.

$$\phi_l = \sum_{j=0}^{n-1} \mathcal{P}_{li} \phi(x_i)$$

where $\mathcal{P}_{li} = w_i P_l(x_i)$. To recover the values at the collocation points we do

$$\phi(x_i) = \sum_{l=0}^{n-1} \mathcal{P}^{-1}_{il} \phi_l$$

where $\mathcal{P}^{-1}_{il} = P_l(x_i)$ is the matrix inverse of $\mathcal{P}_{li}$.

5.1.3. Multi-domain

One of the main drawbacks of pseudospectral collocation methods is that they do not deal correctly with non-regular functions. If the function that we want approximate has discontinuities, even in its first derivatives, the exponential convergence is lost and the approximated function can show oscillations around the discontinuity. This is known as the Gibbs phenomenon.

There are multiple ways to deal with this problem, one of them is to use a multi-domain approach. It consists in dividing the integration domain in multiple subintervals. A division is placed at the points where there is a discontinuity in the function. So now the function is continuous in each subinterval and the pseudospectral approximation works properly.

5.1.4. Numerical differentiation in ESTER

At the moment, ESTER provides two classes for numerical differentiation:

- `diff_gl`: Multi-domain Gauss-Lobatto numerical differentiation.
- `diff_leg`: Gauss-Legendre numerical differentiation for axisymmetric functions on the sphere with a defined type of symmetry (pole,equator).

The function prototypes are defined in `numdiff.h`.

5.2. Multi-domain Gauss-Lobatto numerical differentiation

In the Gauss-Lobatto (or more properly Gauss-Lobatto-Chebyshev) collocation method, the basis functions are Chebyshev polynomials of the first kind

$$T_l(x) = \cos(l \arccos(x))$$

defined in the interval $(-1, 1)$. The collocation points are

$$x_i = -\cos(\frac{i\pi}{n})$$

The end points of the interval are also collocation points, which make this method well-suited for boundary value problems.

In the ESTER library, multi-domain Gauss-Lobatto numerical differentiation is implemented in the `diff_gl` class. To work with this class we should first create an object using

```
diff_gl gl(n);
```

The argument n is optional (default 1) and indicates the number of domains. To change the number of domains we can do

```
gl.set_ndomains(n);
```

After setting the number of domains we must indicate the number of points per domain and the position of the domains. Let's see an example using three domains and the following set-up.

- First domain in the interval (0,5) with 30 points.
- Second domain in the interval (5,7.5) with 20 points.
- Third domain in the interval (7.5,10) with 20 points.

The `diff_gl` object can be initialized using the code

```
diff_gl gl;

gl.set_ndomains(3); // Use 3 domains
gl.set_xif(0.,5.,7.5,10.); // Set the limits between domains
gl.set_npts(30,20,20); // Set the number of points in each domain

gl.init(); // Initialize the object
```

The limits between the subdomains and the number of points are stored in C arrays that are accessible from outside the class, so the code above is equivalent to

```
diff_gl gl;

gl.set_ndomains(3);
gl.xif[0]=0;gl.xif[1]=5;gl.xif[2]=7.5;gl.xif[3]=10;
gl.npts[0]=30;gl.npts[1]=20;gl.npts[2]=20;

gl.init();
```

During the initialization, the following objects are created:

Name	Type	Size	Description
`ndomains`	`int`		Number of domains
`N`	`int`		Total number of points
`x`	`matrix`	(N,1)	Collocation points x_i
`D`	`matrix_block_diag`	(N,N) in blocks of (npts[i],npts[i])	Differentiation matrix
`I`	`matrix`	(1,N)	Integration matrix
`P`	`matrix_block_diag`	(N,N) in blocks of (npts[i],npts[i])	Projection matrix $\mathcal{P}_{ij}$
`P1`	`matrix_block_diag`	(N,N) in blocks of (npts[i],npts[i])	Inverse projection matrix $\mathcal{P}_{ij}^{-1}$

The following example illustrates the use of these objects:

```
[...] // Initialization (previous example)

matrix y,dy,ddy;

y=cos(gl.x); // Definition of function y
dy=(gl.D,y); // First derivative
```

```
ddy=(gl.D,gl.D,y); // Second derivative

double integral;

integral=(gl.I,y)(0); // integral= ∫_{xif[0]=0}^{xif[3]=10} y(x)dx
                          // Note the (0) at the end to convert the result
                          // from matrix(1,1) to double
matrix yl;

yl=(gl.P,y); // Spectral coefficients
y=(gl.P1,yl); // Recover function values from spectral coefficients
```

A function can also be interpolated at any point within the whole domain using the method

```
eval(y,x)
```

where `x` can be of type `double` or `matrix` and the return value will be of always of type `matrix`. It returns the value of `y` at the point(s) `x`. We can use also a third argument of type `matrix`

```
eval(y,x,T)
```

where `T` is modified during the call and can be used to interpolate other functions. For the previous example

```
[...]

double y0,dy0;
matrix T;

y0=gl.eval(y,2.5,T)(0); // Get the value of y in x=2.5.
                          //The (0) at the end is needed because
                          // the result will be a matrix of size (1,1),
                          //and we want the first (and only)
                          // value of this matrix

dy0=(T,dy)(0); // We use the matrix T to calculate the value of dy in x=2.5
```

A `diff_gl` object can be copied using the assignment operator. We can do

```
diff_gl gl1,gl2;

gl1.set_ndomains(2);
gl1.set_npts(10,10);
gl1.set_xif(0.,0.5,1.);
gl2=gl1;
```

Now `gl2` is an independent copy of `gl1` and, as `gl1` has already been initialized, this is also the case of `gl2`.

5.2.1. Example

Let's see a full featured example that uses the class `diff_gl`. We are going to solve the ordinary differential equation

$$x^2\frac{\mathrm{d}^2y}{\mathrm{d}x^2}+x\frac{\mathrm{d}y}{\mathrm{d}x}-y=x^2$$

within the interval $[0,1]$ with boundary conditions

$$y(0)=0 \qquad y(1)=0$$

whose exact solution is

$$y=\frac{1}{3}x(x-1)$$

To simplify the task, we will use only 1 domain. Later, we will see the class `solver` that allows to solve more complicated problems using several domains and several variables.

The code is the following

```
// The following example solves the differential equation
//        x^2*y'' + x*y' - y = x^2
// with boundary conditions y(0)=0 and y(1)=0
// whose exact solution is
//      y = x*(x-1)/3

#include<stdio.h>
#include"numdiff.h"

int main() {

    //Initialize a diff_gl object with 1 domain

    int n=20; // Number of points.
        //In fact, this example can be solved using only 3 points.
    diff_gl gl(1);

    gl.set_npts(n);
    gl.set_xif(0.,1.);
    gl.init();

    // We will work with only 1 domain, so we create a reference to the
    // first (and only) block of gl.D

    matrix &D=gl.D.block(0);
    matrix &x=gl.x;

    // Set up the operator matrix and the right hand side

    matrix op,rhs;

    op=x*x*(D,D)+x*D-eye(n);
    rhs=x*x;
```

```
    // Introduce boundary conditions

    op.setrow(0,zeros(1,n));op(0,0)=1;
    rhs(0)=0;
    op.setrow(-1,zeros(1,n));op(-1,-1)=1;
    rhs(-1)=0;

    // Solve the system

    matrix y;

    y=op.solve(rhs);

    // Interpolate the solution into a finer grid

    matrix x_fine,y_fine;

    x_fine=vector_t(0,1,100);
    y_fine=gl.eval(y,x_fine);

    // Compare with the exact solution

    matrix y_exact;

    y_exact=x_fine*(x_fine-1)/3;

    printf("Solved using %d points \n",gl.N);
    printf("Max. error=%e \n",max(abs(y_fine-y_exact)));

    return 0;

}
```

To run the example, just copy the code above to a file, and compile with `ester_build`. If the file is called `example.cpp` we will do

```
$ ester_build example.cpp -o example
```

and then execute using

```
$ ./example
```

The output will be something like

```
Solved using 20 points
Max. error=5.329938e-16
```

5.3. Gauss-Legendre numerical differentiation

The Gauss-Legendre collocation method uses Legendre polynomials $P_l(x)$ as basis functions. For n points, the collocation points are defined as the roots of $P_n(x)$.

Legendre collocation is particularly adapted to deal with axisymmetric functions on the surface of a sphere, that depend only on the colatitude θ, just by doing $x = \cos\theta$.

The current implementation in ESTER considers only one domain, limited to one hemisphere $\theta \in [0, \pi/2]$. This is more efficient when dealing with functions that have a defined type of symmetry with respect to the equator ($\theta = \pi/2$), which is the case for all of the variables used in the ESTER code. Legendre polynomials $P_l(\cos\theta)$ are symmetric with respect to $\theta = 0$ (the pole). When dealing with antisymmetric functions with respect to the pole, the derivatives $\frac{\mathrm{d}P_l}{\mathrm{d}\theta}$ are used as basis functions instead.

The implementation considers four types of symmetry. Each type is indicated by its own suffix.

Suffix	Pole	Equator	Basis functions
00	Symmetric	Symmetric	$P_l(\cos\theta)$ with l even
01	Symmetric	Antisymmetric	$P_l(\cos\theta)$ with l odd
10	Antisymmetric	Symmetric	$\frac{\mathrm{d}P_l}{\mathrm{d}\theta}$ with l odd
11	Antisymmetric	Antisymmetric	$\frac{\mathrm{d}P_l}{\mathrm{d}\theta}$ with l even

Legendre numerical differentiation is implemented in the class `diff_leg`. In order to use it, we must start by creating an object

```
diff_leg leg;
```

Then we set the number of points by setting the variable `npts`, for example

```
leg.npts=20;
```

and initialize the object

```
leg.init();
```

The following objects are created:

Name	Type	Size	Description
`th`	`matrix`	(1,`npts`)	Collocation points x_i
`D_00, D_01, D_10, D_11`	`matrix`	(`npts`,`npts`)	Differentiation matrices
`D2_00, D2_01, D2_10, D2_11`	`matrix`	(`npts`,`npts`)	Second differentiation matrices
`lap_00, lap_01, lap_10, lap_11`	`matrix`	(`npts`,`npts`)	Laplacian operator $\frac{1}{\sin\theta}\frac{\mathrm{d}}{\mathrm{d}\theta}\left(\sin\theta\frac{\mathrm{d}}{\mathrm{d}\theta}\right)$
`I_00`	`matrix`	(`npts`,1)	Integration matrix
`P_00, P_01, P_10, P_11`	`matrix`	(`npts`,`npts`)	Projection matrices $\mathcal{P}_{ij}$
`P1_00, P1_01, P1_10, P1_11`	`matrix`	(`npts`,`npts`)	Inverse projection matrices $\mathcal{P}_{ij}^{-1}$

Contrary to the `diff_gl` class, in which the functions was supposed to be column vectors, the `diff_leg` class expects functions to be defined as row vectors. This means that all the operators are applied using right multiplication. For example, the derivative will be

```
dy=(y,D_00);
```

The reason for that is more clear when working with 2D functions. Consider the code

```
int nr=30;
int nth=20;

// Inititalize a diff_gl object with nr points
diff_gl gl(1);
gl.set_xif(0.,1.);gl.set_npts(nr);
gl.init();

// Initialize a diff_leg object with nth points
diff_leg leg;
leg.npts=nth;
leg.init();

// Define a 2D function
matrix y;
y=gl.x*sin(leg.th)*sin(leg.th);
    // gl.x is (nr,1) and leg.th is (1,nth), then y is (nr,nth)

//Compute derivatives
matrix dy_x,dy_th,dy_x_th;
dy_x=(gl.D,y); // Derivative with respect to x
dy_th=(y,leg.D_00); // Derivative with respect to th
dy_x_th=(gl.D,y,leg.D_00); // Second derivative with respect to x and th
```

Note that the derivative of a function will have a different type of symmetry. For example, for a symmetric-symmetric 00 function, the first derivative is

```
dy=(y,leg.D_00)
```

which is of type 11. Then to calculate the second derivative we should do

```
ddy=(y,leg.D_00,leg.D_11)
```

or, using the second derivative matrix

```
ddy=(y,leg.D2_00)
```

where `ddy` has type 00.

The integration matrix is defined only for type 00 functions and computes the integral between 0 and π with weight function $\sin\theta$

$$\texttt{(y,leg.I\_00)} = \int_0^\pi y \sin\theta \mathrm{d}\theta$$

We can interpolate functions at any point using `eval_xx`, where `xx` is the type of symmetry, for example

```
eval_00(y,th)
```

gives the value of `y` at the point(s) `th`. Here, `th` can be either `double` or `int` and the returned value is always of type `matrix`. We may also use a third argument

```
eval_00(y,th,T)
```

where T can be used to interpolate additional functions at the same point(s) by doing

```
(y2,T)
```

A diff_leg object can be copied using the assignment operator. We can do

```
diff_leg leg1,leg2;

leg1.npts=20;leg1.init();
leg2=leg1;
```

Now leg2 is an independent copy of leg1.

5.3.1. Example

Let's see a more complete example. We will consider axisymmetric functions in spherical coordinates, that is functions that depend only on r and θ. For the radial direction we use Gauss-Lobatto differentiation in the interval $(0,1)$, and Legendre differentiation for the latitudinal direction. We will write two functions, one for calculating the value of the laplacian at a certain point and another one to evaluate the volume integral within the whole sphere. For simplicity, we will consider only type 00 functions.

The code is as follows

```
/* The following example illustrates the use of the numerical differentiation
library in 2D in spherical coordinates */

#include<stdio.h>
#include"numdiff.h"
#include"constants.h" //For the defintion of PI

//Function prototypes
double laplacian(matrix y,double r0,double th0);
double integral(matrix y);

// Define diff_gl and diff_leg objects as global variables
diff_gl gl;
diff_leg leg;
// Create references for spherical coordinates
matrix &r=gl.x,&th=leg.th;

int main() {

    //Initialize gl. In the example we will use 2 domains
    gl.set_ndomains(2);
    gl.set_xif(1e-3,0.2,1.); // Use 1e-3 as the interior limit (instead of 0)
                            // to avoid a division by zero in the
                            // calculation of the laplacian
    gl.set_npts(100,100);
    gl.init();
    //Initialize leg
```

```
    leg.npts=50;
    leg.init();

    matrix y;

    //Define the function y
    y=r*r*r*(1+sin(th)*sin(th));

    double lap_y,int_y;
    double r0=0.3,th0=PI/3;

    lap_y=laplacian(y,r0,th0);
    int_y=integral(y);

    printf("The value of the laplacian at (%f,%f) is %e \n",r0,th0,lap_y);
    printf("The volume integral is %e \n",int_y);

    return 0;

}

// Function for calculating the laplacian of y at (r0,th0)
double laplacian(matrix y,double r0,double th0) {

    matrix lap_y;

    lap_y=(gl.D,r*r*gl.D,y)/r/r+(y,leg.lap_00)/r/r;

    //Interpolate in the direction of r
    lap_y=gl.eval(lap_y,r0); // Now lap_y is (1,nth)
    //Interpolate in the direction of theta
    lap_y=leg.eval_00(lap_y,th0); // Now lap_y is (1,1)

    return lap_y(0);
    // lap_y has only 1 element, but we must include (0)
    // at the end to return a double
}
// Function for calculating the volume integral of y
double integral(matrix y) {

    return 2*PI*(gl.I,r*r*y,leg.I_00)(0);

}
```

After running the code, the output should be

```
The value of the laplacian at (0.300000,1.047198) is 6.150000e+00
The volume integral is 3.490659e+00
```

5.4. Reference

Gauss-Lobatto differentiation

Data members

ndomains	P	xif
N	P1	
x	I	
D	npts	

Functions

set_ndomains(n)	set_xif(x0,x1,...)	eval(y,x,T)
set_npts(n0,n1,...)	init()	

Legendre differentiation

Data members

npts	P1_00,P1_01,P1_10,P1_11	lap_00,lap_01,lap_10,lap_11
th	D_00,D_01,D_10,D_11	I_00
P_00,P_01,P_10,P_11	D2_00,D2_01,D2_10,D2_11	

Functions

init()	eval_10(y,th,T)
eval_00(y,th,T)	eval_11(y,th,T)
eval_01(y,th,T)	eval(y,th,T,par_pol,par_eq)

5.4.1. Gauss-Lobatto differentiation

Data members

---

ndomains

Type: int

Description: Number of domains (read-only).

N

Type: int

Description: Total number of points, including all the domains (read-only).

---

x

Type: matrix

Description: Collocation points (N x 1).

D

Type:	matrix_block_diag
Description:	Differentiation matrix.

P

Type:	matrix_block_diag
Description:	Projection matrix.

P1

Type:	matrix_block_diag
Description:	Inverse projection matrix.

I

Type:	matrix
Description:	Integration matrix.

npts

Type:	int *
Description:	Array of size ndomains() with the number of points in each domain.

xif

Type:	double *
Description:	Array of size ndomains()+1 with the limits between domains.

Functions

set_ndomains(n)

Type:	Method
Inputs:	n (int): Number of domains
Output:	None
Description:	Change the number of domains.

set_npts(n0,n1,...)

Type:	Method
Inputs:	n0,n1,... (int): Number of points
Output:	None
Description:	Change the number of points in each domain.

set_xif(x0,x1,...)

Type: Method
Inputs: n0,n1,... (double): Number of points
Output: None
Description: Change the position of the limits between domains.

init()

Type: Method
Inputs: None
Output: None
Description: Initializes the object.

eval(y,x,T)

Type: Method
Inputs: y (matrix): Function to evaluate
x (matrix or double): Evaluation point(s)
T (matrix): Interpolating matrix (optional, output)

Output: matrix
Description: Evaluate function y at point(s) x. If y is NxM and x is Kx1, the returned matrix is KxM. The optional matrix T can be used to interpolate additional functions by multiplying (T,y2).

5.4.2. Legendre differentiation

Data members

npts

Type: int
Description: Number of points.

th

Type: matrix
Description: θ values of the collocation points (1 x npts).

P_00,P_01,P_10,P_11

Type: matrix
Description: Projection matrices for each type of symmetry.

P1_00,P1_01,P1_10,P1_11

Type: matrix
Description: Inverse projection matrices for each type of symmetry.

D_00,D_01,D_10,D_11

Type: matrix
Description: Differentiation matrices for each type of symmetry.

D2_00,D2_01,D2_10,D2_11

Type: matrix
Description: Second differentiation matrices for each type of symmetry.

lap_00,lap_01,lap_10,lap_11

Type: matrix
Description: Laplacian operator matrices for each type of symmetry.

$$(\texttt{lap\_xx}, \texttt{f}) \equiv \frac{1}{\sin\theta}\frac{\mathrm{d}}{\mathrm{d}\theta}\left(\sin\theta\frac{\mathrm{d}\texttt{f}}{\mathrm{d}\theta}\right)$$

I_00

Type: matrix
Description: Integration matrix (npts x 1) for symmetric functions.

$$(\texttt{I\_00}, \texttt{f}) \equiv \int_0^\pi \texttt{f}\sin(\theta)\mathrm{d}\theta$$

Functions

init()

Type: Method
Inputs: None
Output: None
Description: Initializes the object.

eval_00(y,th,T)

Type: Method

Inputs: y (matrix): Function to evaluate
th (matrix or double): Evaluation point(s)
T (matrix): Interpolating matrix (optional, output)

Output: matrix

Description: Evaluate function y(θ) at point(s) th. The function y should be symmetric at the pole and the equator. If y is MxN and x is 1xK, the returned matrix is MxK. The optional matrix T can be used to interpolate additional functions by multiplying (T,y2).

---

eval_01(y,th,T)

Type: Method

Inputs: y (matrix): Function to evaluate
th (matrix or double): Evaluation point(s)
T (matrix): Interpolating matrix (optional, output)

Output: matrix

Description: Evaluate function y(θ) at point(s) th. The function y should be symmetric at the pole and antisymmetric the equator. If y is MxN and x is 1xK, the returned matrix is MxK. The optional matrix T can be used to interpolate additional functions by multiplying (T,y2).

---

eval_10(y,th,T)

Type: Method

Inputs: y (matrix): Function to evaluate
th (matrix or double): Evaluation point(s)
T (matrix): Interpolating matrix (optional, output)

Output: matrix

Description: Evaluate function y(θ) at point(s) th. The function y should be antisymmetric at the pole and symmetric at the equator. If y is MxN and x is 1xK, the returned matrix is MxK. The optional matrix T can be used to interpolate additional functions by multiplying (T,y2).

eval_11(y,th,T)

Type: Method

Inputs: y (matrix): Function to evaluate
th (matrix or double): Evaluation point(s)
T (matrix): Interpolating matrix (optional, output)

Output: matrix

Description: Evaluate function y(θ) at point(s) th. The function y should be antisymmetric at the pole and the equator. If y is MxN and x is 1xK, the returned matrix is MxK. The optional matrix T can be used to interpolate additional functions by multiplying (T,y2).

---

eval(y,th,T,par_pol,par_eq)

Type: Method

Inputs: y (matrix): Function to evaluate
th (matrix or double): Evaluation point(s)
T (matrix): Interpolating matrix (output)
par_pol (int): Type of symmetry at the pole
par_eq (int): Type of symmetry at the equator

Output: matrix

Description: Evaluate function y(θ) at point(s) th. par_pol and par_eq can be 0 (symmetric) or 1 (antisymmetric). If y is MxN and x is 1xK, the returned matrix is MxK. The matrix T can be used to interpolate additional functions by multiplying (T,y2).

6 Mapping. Spheroidal coordinates

We have seen at the end of the previous chapter that we can work with 2D problems combining `diff_gl` and `diff_leg` objects. This is exactly the purpose of the `mapping` class, that contains all the elements to work with spherical, and more important, with deformed spheroidal domains.

6.1. Introduction

Rotating stars are not spherical. The centrifugal force will flatten the star, and this flattening will be more important as the rotation velocity increases. For this reason, we have to find an appropriate way to deal with the discretization of the stellar variables in a deformed spheroidal domain.

Note that the problems in which we are interested will involve only axisymmetric quantities, thus essentially 2D. For that reason, we will restrict the discussion to axisymmetric 2D problems in a spherical-like domain, but it could be also generalized to other geometries.

6.1.1. Coordinate mapping

We will consider spherical coordinates (r, θ, φ). Let $\mathcal{D}$ be an axisymmetric domain centered in the origin of coordinates ($r = 0$) and whose outer boundary $\partial\mathcal{D}$ can be represented by a function $R(\theta)$ that depends only on the colatitude. We define a new set of coordinates $(\zeta, \theta', \varphi')$ such that the new radial-like coordinate ζ is constant over $\partial\mathcal{D}$. This new spheroidal coordinates are defined by the transformation:

$$\begin{cases} r = r(\zeta, \theta') \\ \theta = \theta' \\ \varphi = \varphi' \end{cases} \tag{6.1}$$

The problem reduces to find a suitable form for the function $r(\zeta, \theta)$.

In our case we are going a little bit further. We will split the domain $\mathcal{D}$ in n subdomains $\mathcal{D}_i$ with $i = 0, \ldots, n-1$. The frontiers between this subdomains are represented by a series of functions $R_i(\theta)$, $i = 0, \ldots, n$, such that the $\mathcal{D}_i \in [R_i(\theta), R_{i+1}(\theta)]$. Note that $R_n(\theta) = R(\theta)$ is the outer boundary of the whole domain and, if the domain contains the origin of coordinates $R_0(\theta) = 0$.

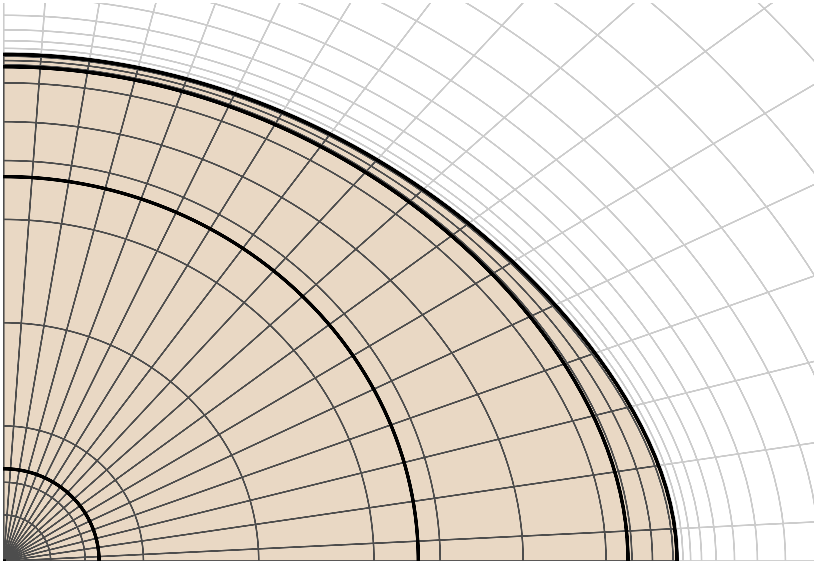


Figure 6.1: Coordinate mapping.

We will have also an external domain $\mathcal{D}_{ex}$ that extends from the outer boundary $R_n(\theta)$ to infinity. It will be useful for writing boundary conditions for certain variables as, for example, the gravitational potential.

We will use a technique adopted from Bonazzola *et al.* (1998), in each subdomain $\mathcal{D}_i$ we use a mapping in the form

$$r(\zeta,\theta) = a_i\xi\Delta\eta_i + R_i(\theta) + A_i(\xi)(\Delta R_i(\theta) - a_i\Delta\eta_i) \qquad \text{for } \zeta \in [\eta_i, \eta_{i+1}] \tag{6.2}$$

where we have defined:

$$\begin{aligned} \eta_i &= R_i(\theta = 0) \\ \Delta\eta_i &= \eta_{i+1} - \eta_i \\ \Delta R_i(\theta) &= R_{i+1}(\theta) - R_i(\theta) \\ \xi &= \frac{\zeta - \eta_i}{\Delta\eta_i} \end{aligned}$$

The function(s) $A_i(\xi)$ and the constant(s) a_i will determine the final form of the mapping. In particular, the function $A_i(\xi)$ should verify the following conditions:

$$\begin{aligned} r(\eta_i,\theta) = R(\theta) \quad &\longrightarrow \quad A_i(\xi = 0) = 0 \\ r(\eta_{i+1},\theta) = R_{i+1}(\theta) \quad &\longrightarrow \quad A_i(\xi = 1) = 1 \end{aligned}$$

The simplest possibility is what we will call a linear mapping

$$A_i(\xi) = \xi \qquad a_i = 0 \tag{6.3}$$

that gives

$$r(\zeta,\theta) = R_i(\theta) + \xi\Delta R_i(\theta) \tag{6.4}$$

In Bonazzola *et al.* (1998), the mapping proposed by the authors satisfies some extra conditions to make it suitable for spectral methods. For that reason they set $A_i'(\xi=0)=0$ and $A_i'(\xi=1)=0$. Doing this, the first derivative of the mapping

$$r_\zeta = \frac{\partial r}{\partial \zeta} = a_i + A_i'(\xi)\left(\frac{\Delta R_i(\theta)}{\Delta \eta_i} - a_i\right) \tag{6.5}$$

is constant over the boundaries of the domains. This facilitates the writing of interface conditions for the derivatives of the variables in the problem when they are expressed in terms of their spectral coefficients. In this case, we will set

$$A_i(\xi) = -2\xi^3 + 3\xi^2 \qquad \text{for } i = 1,\dots,n-1 \tag{6.6}$$

$$A_0(\xi) = -1.5\xi^5 + 2.5\xi^3 \tag{6.7}$$

The constant a_i deserves some special attention. We want $r_\zeta > 0$, that is, r being monotonically increasing with ζ, then a_i should satisfy the condition

$$a_i(A_i'(\xi) - 1) < A_i'(\xi)\frac{\Delta R_i(\theta)}{\Delta \eta_i} \tag{6.8}$$

For $A_i'(\xi) = 0$ (the boundaries), the condition states $a_i > 0$. When $A'(\xi) < 1$, the condition is automatically satisfied (note that $A_i'(\xi)$ is always positive). But, as $A(\xi)$ should go from 0 at $\xi = 0$ to 1 at $\xi = 1$, we know that $\max(A_i'(\xi)) \geq 1$, where the equality corresponds to the linear mapping that we have seen before. So, in the worst case, the condition becomes

$$a_i < \frac{1}{1 - 1/\max(A_i'(\xi))}\frac{\min(\Delta R_i(\theta))}{\Delta \eta_i} \tag{6.9}$$

or, using (6.6)

$$\begin{aligned} a_i &< 3\frac{\min(\Delta R_i(\theta))}{\Delta \eta_i} \qquad \text{for } i = 1,\dots,n-1 \\ a_0 &< \frac{15}{7}\frac{\min(\Delta R_0(\theta))}{\Delta \eta_0} \end{aligned} \tag{6.10}$$

In practice, we will take $a_i = 1$. This is motivated by the fact that we will work with oblate domains, and the flattening will increase for the successive subdomains, then $\min(\Delta R_i(\theta)) = \Delta\eta_i$, and the conditions are fully satisfied. Also, working with a fixed value of a_i makes easier to work with problems where the frontiers between the subdomains are not known a priori, and the jacobian of the mapping becomes a smooth function suitable for iterative methods.

The general form of the jacobian of the mapping is defined by the expression

$$\delta r^i = J_0^{(i)}(\zeta,\theta)\delta\eta_i + J_1^{(i)}(\zeta,\theta)\delta\Delta\eta_i + J_2^{(i)}(\zeta,\theta)\delta R_i(\theta) + J_3^{(i)}(\zeta,\theta)\delta\Delta R_i(\theta) \tag{6.11}$$

and using (6.2), for fixed a_i

$$\begin{aligned} J_0^{(i)} &= 0 \\ J_1^{(i)} &= a_i(\xi - A_i(\xi)) \\ J_2^{(i)} &= 1 \\ J_3^{(i)} &= A_i(\xi) \end{aligned} \tag{6.12}$$

For the external domain, we will take

$$\xi_{ex} = \frac{\zeta - \eta_n}{\eta_n} \qquad \xi \in [0,\infty) \tag{6.13}$$

and

$$r_{ex}(\zeta,\theta) = \xi + R(\theta) \tag{6.14}$$

with jacobian

$$\begin{aligned} J_0^{(ex)} &= 0 \\ J_1^{(ex)} &= 0 \\ J_2^{(ex)} &= 1 \\ J_3^{(ex)} &= 0 \end{aligned} \tag{6.15}$$

6.1.2. Spheroidal coordinates

In the previous section, we have defined a system of spheroidal coordinates (ζ,θ',φ'), where $\theta' = \theta$ and $\varphi' = \varphi$ correspond to the usual spherical coordinates and ζ is defined by a relation $r = r(\zeta, \theta)$. These spheroidal coordinates are non-orthogonal, which means that the surfaces of constant ζ are not perpendicular to those of constant θ.

Before we continue, we should clarify a point. We have set $\theta' = \theta$, so hereafter we will remove the tilde ($'$) to simplify the notation. But when working with spheroidal coordinates, the partial derivative $\frac{\partial}{\partial\theta}$ refers to the derivative with respect to θ with ζ constant, that is not the habitual derivative in spherical coordinates that is done holding r constant.

$$\left.\frac{\partial}{\partial\theta}\right|_{\zeta,\varphi\,\text{const.}} \neq \left.\frac{\partial}{\partial\theta}\right|_{r,\varphi\,\text{const.}}$$

The same can be said for the azimuthal coordinate φ but, in this case $\left.\frac{\partial}{\partial\varphi}\right|_{\zeta,\theta\,\text{const.}} = \left.\frac{\partial}{\partial\varphi}\right|_{r,\theta\,\text{const.}}$.

Natural basis

We will start by defining the natural basis for the spheroidal coordinates, we have two sets of basis vectors:

- Covariant basis vectors: $\mathbf{E}_i = \dfrac{\partial \mathbf{r}}{\partial x^i}$

$$\mathbf{E}_\zeta = r_\zeta \hat{\boldsymbol{r}}, \quad \mathbf{E}_\theta = r_\theta \hat{\boldsymbol{r}} + r\hat{\boldsymbol{\theta}}, \quad \mathbf{E}_\varphi = r\sin\theta \hat{\boldsymbol{\varphi}}, \tag{6.16}$$

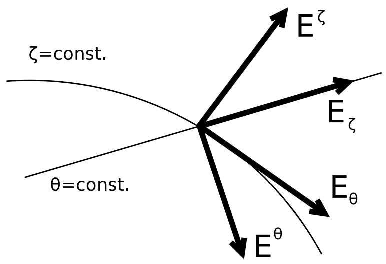
- Contravariant basis vectors: $\mathbf{E}^i = \nabla x^i$

$$\mathbf{E}^\zeta = \frac{\hat{\boldsymbol{r}}}{r_\zeta} - \frac{r_\theta}{r r_\zeta}\hat{\boldsymbol{\theta}}, \quad \mathbf{E}^\theta = \frac{\hat{\boldsymbol{\theta}}}{r}, \quad \mathbf{E}^\varphi = \frac{\hat{\boldsymbol{\varphi}}}{r\sin\theta}, \tag{6.17}$$

where $\hat{\boldsymbol{r}}, \hat{\boldsymbol{\theta}}, \hat{\boldsymbol{\varphi}}$ are the usual unit vectors in spherical coordinates, and

$$r_\zeta = \frac{\partial r}{\partial \zeta} \quad r_\theta = \frac{\partial r}{\partial \theta}$$

The vectors of the natural basis are not unit vectors. The covariant vector $\mathbf{E}_i$ is parallel to the line $x^j =$ const. with $j \neq i$, while the contravariant vector $\mathbf{E}^i$ is perpendicular to the surface $x^i =$ const. For orthogonal coordinates $\mathbf{E}_i \parallel \mathbf{E}^i$, but this is not the case for non-orthogonal coordinates.



The basis vectors satisfy

$$\mathbf{E}_i \cdot \mathbf{E}^j = \mathbf{E}^i \cdot \mathbf{E}_j = \delta_{ij} \tag{6.18}$$

where δ_{ij} is the Kronecker's delta.

Using the basis vectors, we can calculate the metric tensor

$$g_{ij} = \mathbf{E}_i \cdot \mathbf{E}_j = \begin{pmatrix} r_\zeta^2 & r_\zeta r_\theta & 0 \\ r_\zeta r_\theta & r^2 + r_\theta^2 & 0 \\ 0 & 0 & r^2 \sin^2\theta \end{pmatrix} \tag{6.19}$$

or, in contravariant form

$$g^{ij} = \mathbf{E}^i \cdot \mathbf{E}^j = \begin{pmatrix} \dfrac{r^2 + r_\theta^2}{r^2 r_\zeta^2} & \dfrac{-r_\theta}{r^2 r_\zeta} & 0 \\ \dfrac{-r_\theta}{r^2 r_\zeta} & \dfrac{1}{r^2} & 0 \\ 0 & 0 & \dfrac{1}{r^2 \sin^2\theta} \end{pmatrix} \tag{6.20}$$

Note that g^{ij} is the matrix inverse of g_{ij}

$$g_{ij} g^{jk} = \delta_{ij}$$

where we have used the Einstein's summation convention, that implies summation over repeated indices.

Given two points x^i and $x^i + \mathrm{d}x^i$, the distance ($\mathrm{d}s$) between them is given by the metric tensor:

$$\mathrm{d}s^2 = g_{ij}\mathrm{d}x^i \mathrm{d}x^j = r_\zeta^2 \mathrm{d}\zeta^2 + 2 r_\zeta r_\theta \mathrm{d}\zeta \mathrm{d}\theta + (r^2 + r_\theta^2)\mathrm{d}\theta^2 + r^2 \sin^2\theta \mathrm{d}\varphi^2 \tag{6.21}$$

The basis vectors verify

$$\mathbf{E}_i \cdot (\mathbf{E}_j \times \mathbf{E}_k) = \epsilon_{ijk} \tag{6.22}$$

and

$$\mathbf{E}^i \cdot (\mathbf{E}^j \times \mathbf{E}^k) = \epsilon^{ijk} \tag{6.23}$$

where ϵ^{ijk} is the Levi-Civita tensor

$$\epsilon_{ijk} = \sqrt{|g|}[i,j,k] \tag{6.24}$$

$$\epsilon^{ijk} = \frac{1}{\sqrt{|g|}}[i,j,k] \tag{6.25}$$

where $|g| = \det(g_{ij}) = r^4 r_\zeta^2 \sin^2\theta$ and

$$[i,j,k] = \begin{cases} 1 & \text{the arguments are an even permutation} \\ -1 & \text{the arguments are an odd permutation} \\ 0 & \text{two or more arguments are equal} \end{cases} \tag{6.26}$$

Representation of vectors

A vector $\mathbf{v}$ can be represented either in covariant or contravariant form:

- Covariant form: $\mathbf{v} = V_\zeta \mathbf{E}^\zeta + V_\theta \mathbf{E}^\theta + V_\varphi \mathbf{E}^\varphi$
- Contravariant form: $\mathbf{v} = V^\zeta \mathbf{E}_\zeta + V^\theta \mathbf{E}_\theta + V^\varphi \mathbf{E}_\varphi$

Here, V_i are the covariant components of the vector $\vec{v}$ and V^i the contravariant components. Note that

$$\mathbf{E}_i \cdot \mathbf{v} = V_i \qquad \text{and} \qquad \mathbf{E}^i \cdot \mathbf{v} = V^i$$

We can use the metric tensor to pass from one representation to the other, indeed

$$V_i = \mathbf{E}_i \cdot \mathbf{v} = \mathbf{E}_i \cdot (\mathbf{E}_j V^j) = g_{ij} V^j \tag{6.27}$$

and similarly

$$V^i = g^{ij} V_j \tag{6.28}$$

Let $(v_r, v_\theta, v_\varphi)$ be the spherical components of a vector $\mathbf{v}$ such that $\mathbf{v} = v_r \hat{\boldsymbol{r}} + v_\theta \hat{\boldsymbol{\theta}} + v_\varphi \hat{\boldsymbol{\varphi}}$. Its spheroidal components will be

$$V_\zeta = r_\zeta v_r, \quad V_\theta = r_\theta v_r + r v_\theta, \quad V_\varphi = r \sin\theta v_\varphi \tag{6.29}$$

and

$$V^\zeta = \frac{v_r}{r_\zeta} - \frac{r_\theta}{r r_\zeta} v_\theta, \quad V^\theta = \frac{v_\theta}{r}, \quad V^\varphi = \frac{v_\varphi}{r \sin\theta} \tag{6.30}$$

We can see from this expressions that V^θ an V^φ are in fact angular velocities.

Using the properties of the basis vectors it can be shown that the scalar product of two vectors is given by

$$\mathbf{a} \cdot \mathbf{b} = A_i B^i = A^i B_i \tag{6.31}$$

and the cross product is

$$\begin{aligned} (\mathbf{a} \times \mathbf{b})^i &= \epsilon^{ijk} A_j B_k \\ (\mathbf{a} \times \mathbf{b})_i &= \epsilon_{ijk} A^j B^k \end{aligned} \tag{6.32}$$

We have presented the basics of the representation of vectors in spheroidal coordinates, let'see now a little example. Consider a surface $\mathcal{S}$ defined by $\zeta = \text{const.}$ as for example the surface of a star or the frontier between two subdomains. We want to calculate the normal and tangential projections of a vector $\mathbf{v}$ with respect to $\mathcal{S}$. First, we define a unit vector $\hat{\mathbf{n}}$, perpendicular to $\mathcal{S}$. For that, we just recall that $\mathbf{E}^\zeta$ is perpendicular to the surfaces $\zeta = \text{const.}$, but it is not a unit vector, so

$$\hat{\mathbf{n}} = \frac{\mathbf{E}^\zeta}{|\mathbf{E}^\zeta|} = \frac{\mathbf{E}^\zeta}{\sqrt{\mathbf{E}^\zeta \cdot \mathbf{E}^\zeta}} = \frac{\mathbf{E}^\zeta}{\sqrt{g^{\zeta\zeta}}} \tag{6.33}$$

then, the normal projection is

$$\hat{\mathbf{n}} \cdot \mathbf{v} = \frac{V^\zeta}{\sqrt{g^{\zeta\zeta}}} = \frac{r_\zeta V^\zeta}{\sqrt{1 + \frac{r_\theta^2}{r^2}}} \tag{6.34}$$

For the parallel projection we have two vectors, the first one, in the direction of φ is just the spherical unit vector $\hat{\boldsymbol{\varphi}}$, in the latitudinal direction, however, it will be

$$\hat{\mathbf{t}} = \frac{\mathbf{E}_\theta}{|\mathbf{E}_\theta|} = \frac{\mathbf{E}_\theta}{\sqrt{\mathbf{E}_\theta \cdot \mathbf{E}_\theta}} = \frac{\mathbf{E}_\theta}{\sqrt{g_{\theta\theta}}} \tag{6.35}$$

so, the parallel projections over $\mathcal{S}$ are

$$\hat{\mathbf{t}} \cdot \mathbf{v} = \frac{V_\theta}{\sqrt{g_{\theta\theta}}} = \frac{1}{\sqrt{1 + \frac{r_\theta^2}{r^2}}} \frac{V_\theta}{r} \tag{6.36}$$

and

$$\hat{\boldsymbol{\varphi}} \cdot \mathbf{v} = \frac{V_\varphi}{r \sin\theta} \tag{6.37}$$

Tensors

A second order tensor $\mathcal{T}$ is represented using 2 indices

$$\mathcal{T} = T^{ij}\mathbf{E}_i\mathbf{E}_j = T_{ij}\mathbf{E}^i\mathbf{E}^j = T^i j\mathbf{E}_i\mathbf{E}^j = T_i{}^j\mathbf{E}^i\mathbf{E}_j \tag{6.38}$$

Again, we can use the metric tensor to lower and raise indices

$$\begin{array}{l} T^{ij} = g^{ik}T_k{}^j = g^{jl}T^i{}_l = g^{ik}g^{jl}T_{kl} \\ T_{ij} = g_{ik}T^k{}_j = g_{jl}T_i{}^l = g_{ik}g_{jl}T^{kl} \end{array} \tag{6.39}$$

The tensor product of 2 vectors is a tensor

$$(\mathbf{a}\,\mathbf{b})^{ij} = a^i b^j \tag{6.40}$$

The dot product between a tensor and a vector is

$$(\mathcal{T} \cdot \mathbf{v})^i = T^{ij}V_j \tag{6.41}$$

and between a vector and a tensor

$$(\mathbf{v} \cdot \mathcal{T})^j = T^{ij}V_i \tag{6.42}$$

Finally, the double dot product is a scalar

$$\mathcal{T} : \mathcal{T} = T^{ij}T_{ij} \tag{6.43}$$

All of this can be generalized to higher order tensors. Note that vectors are in fact tensors of order 1.

Differential operators

Our goal is to be able to write differential equations using spheroidal coordinates. We will start finding the relation between the partial derivatives with respect to the spherical coordinates and those calculated with respect to the spheroidal coordinates. We will add the primes ($'$) in the notation for the spheroidal θ' and φ' coordinates to clarify the notation, so the derivative with respect to a spheroidal coordinate is done holding the other spheroidal coordinates constant. Following the chain rule

$$\frac{\partial}{\partial r} = \frac{\partial\zeta}{\partial r}\frac{\partial}{\partial\zeta} + \frac{\partial\theta'}{\partial r}\frac{\partial}{\partial\theta'} + \frac{\partial\varphi'}{\partial r}\frac{\partial}{\partial\varphi'} \tag{6.44}$$

Obviously, $\dfrac{\partial\theta'}{\partial r} = \dfrac{\partial\varphi'}{\partial r} = 0$, and

$$\mathrm{d}r = r_\zeta \mathrm{d}\zeta + r_\theta \mathrm{d}\theta$$

$$\mathrm{d}\zeta = \frac{1}{r_\zeta}\mathrm{d}r - \frac{r_\theta}{r_\zeta}\mathrm{d}\theta$$

where we see $\frac{\partial \zeta}{\partial r} = \frac{1}{r_\zeta}$ and $\frac{\partial \zeta}{\partial \theta} = -\frac{r_\theta}{r_\zeta}$. Then

$$\frac{\partial}{\partial r} = \frac{1}{r_\zeta}\frac{\partial}{\partial \zeta} \tag{6.45}$$

The other partial derivatives are calculated in the same way

$$\frac{\partial}{\partial \theta} = \frac{\partial}{\partial \theta'} - \frac{r_\theta}{r_\zeta}\frac{\partial}{\partial \zeta} \tag{6.46}$$

$$\frac{\partial}{\partial \varphi} = \frac{\partial}{\partial \varphi'} \tag{6.47}$$

Of course, we could take this expressions and substitute them into the expressions for the differential operators corresponding to the spherical coordinates, but there is a much more efficient way to do it.

First, let's define the general form of the gradient of a scalar quantity. The gradient will be a vector, whose covariant components are

$$(\nabla\phi)_i = \frac{\partial \phi}{\partial x^i} = \phi_{,i} \tag{6.48}$$

where we have introduced the comma notation for the partial derivative. The contravariant components of the gradient will be

$$(\nabla\phi)^i = g^{ij}\phi_{,j} \tag{6.49}$$

We can also derive a component of a vector V^i in the same way. However, this derivative

$$\frac{\partial V^i}{\partial x^j} = V^i{}_{,j} \tag{6.50}$$

is not a tensor, as it does not transform correctly under a change of coordinates. That's why we will introduce the covariant derivative

$$\nabla_j V^i = V^i{}_{;j} = V^i{}_{,j} + \Gamma^i{}_{kj}V^k \tag{6.51}$$

Where $\Gamma^i{}_{kj} = \mathbf{E}^i \cdot \frac{\partial \mathbf{E}_k}{\partial x^j}$ is a Christoffel symbol of the second kind. The covariant derivative of a vector $V^i{}_{;j}$ is a tensor that represents the gradient of the vector.

$$(\nabla\mathbf{v})^{ij} = g^{jk}(\nabla\mathbf{v})^i{}_k = g^{jk}V^i{}_{;k} \tag{6.52}$$

We can also calculate the covariant derivative using the covariant components of the vector

$$\nabla_j V_i = V_{i;j} = V_{i,j} - \Gamma^k{}_{ij}V_k \tag{6.53}$$

The Christoffel symbols can be calculated using the following relation

$$\Gamma^i_{jk} = \frac{1}{2}g^{il}(g_{lj,k} + g_{lk,j} - g_{jk,l}) \tag{6.54}$$

where we can see that they are symmetric with respect to the second and third indices $\Gamma^i_{jk} = \Gamma^i_{kj}$. They also verify

$$\Gamma^i_{ji} = \frac{\log\sqrt{|g|}}{\partial x^j} \tag{6.55}$$

The covariant derivative of second order tensors is done in a similar way

$$\nabla_k T^{ij} = T^{ij}{}_{;k} = T^{ij}{}_{,k} + \Gamma^i{}_{lk} T^{lj} + \Gamma^j{}_{lk} T^{il} \tag{6.56}$$

If one of the indices is covariant, then we do

$$\nabla_k T^i{}_j = T^i{}_{j;k} = T^i{}_{j,k} + \Gamma^i{}_{lk} T^l{}_j - \Gamma^l{}_{jk} T^i{}_l \tag{6.57}$$

where we can see the general rule valid also for higher order tensors, the covariant derivative is equal to the regular derivative plus:

- For each contravariant index, we add $\Gamma^i{}_{lk} T^{\ldots l \ldots}$
- For each covariant index, we subtract $\Gamma^l{}_{ik} T_{\ldots l \ldots}$

Using the covariant derivative, we can calculate all the differential operators in spheroidal coordinates. We have already see the gradient of a scalar and a vector, similarly, the divergence of a vector will be

$$\nabla \cdot \mathbf{v} = \nabla_i V^i = V^i{}_{;i} \tag{6.58}$$

and for a tensor

$$(\nabla \cdot \mathcal{T})^i = \nabla_j T^{ij} = T^{ij}{}_{;j} \tag{6.59}$$

Note that some authors prefer the definition $(\nabla \cdot \mathcal{T})^j = \nabla_i T^{ij} = T^{ij}{}_{;i}$. Using the expresion for the cross product, we can calculate the curl of a vector

$$(\nabla \times \mathbf{v})^i = \epsilon^{ijk} \nabla_j V_k = \epsilon^{ijk} V_{k;j} \tag{6.60}$$

The laplacian of a scalar field will be

$$\Delta\phi = \nabla \cdot (\nabla\phi) = \nabla_i (g^{ij} \nabla_j \phi) = (g^{ij} \phi_{,j})_{;i} \tag{6.61}$$

and for a vector field

$$(\Delta\mathbf{v})^i = \nabla_j (g^{jk} \nabla_k V^i) = (g^{jk} V^i{}_{;k})_{;j} \tag{6.62}$$

The material derivative is

$$[(\mathbf{v} \cdot \nabla)\mathbf{v}]^i = V^j \nabla_j V^i = V^j V^i{}_{;j} \tag{6.63}$$

Useful relations

- Line, area and volume elements
  - Line element

$$\mathrm{d}s^2 = g_{ij} \mathrm{d}x^i \mathrm{d}x^j = r_\zeta^2 \mathrm{d}\zeta^2 + 2 r_\zeta r_\theta \mathrm{d}\zeta \mathrm{d}\theta + (r^2 + r_\theta^2)\mathrm{d}\theta^2 + r^2 \sin^2\theta \mathrm{d}\varphi^2 \tag{6.64}$$

$$\mathrm{d}\mathbf{r} = \mathbf{E}_i \mathrm{d}x^i = \mathbf{E}_\zeta \mathrm{d}\zeta + \mathbf{E}_\theta \mathrm{d}\theta + \mathbf{E}_\varphi \mathrm{d}\varphi \tag{6.65}$$

– Area element in a surface ζ =const.

$$\mathrm{d}\mathbf{S} = (\mathbf{E}_\theta \times \mathbf{E}_\varphi)\mathrm{d}\theta\mathrm{d}\varphi = r^2 r_\zeta \sin\theta \mathbf{E}^\zeta \mathrm{d}\theta\mathrm{d}\varphi \tag{6.66}$$

$$\mathrm{d}S = |\mathrm{d}\mathbf{S}| = \sqrt{g^{\zeta\zeta}} r^2 r_\zeta \sin\theta \mathrm{d}\theta\mathrm{d}\varphi = r^2 \sqrt{1 + \frac{r_\theta^2}{r^2}} \sin\theta\mathrm{d}\theta\mathrm{d}\varphi \tag{6.67}$$

– Area element in a surface of constant $p = p(\zeta, \theta)$.

$$\mathrm{d}\mathbf{S} = r^2 r_\zeta \sin\theta \left(\mathbf{E}^\zeta + \frac{p_{,\theta}}{p_{,\zeta}}\mathbf{E}^\theta\right) \mathrm{d}\theta\mathrm{d}\varphi \tag{6.68}$$

$$\mathrm{d}S = |\mathrm{d}\mathbf{S}| = r^2 r_\zeta \sin\theta \sqrt{g^{\zeta\zeta} + 2\frac{p_{,\theta}}{p_{,\zeta}} g^{\zeta\theta} + \left(\frac{p_{,\theta}}{p_{,\zeta}}\right)^2 g^{\theta\theta}} \mathrm{d}\theta\mathrm{d}\varphi \tag{6.69}$$

– Volume element

$$\mathrm{d}V = \mathbf{E}_\zeta \cdot (\mathbf{E}_\theta \times \mathbf{E}_\varphi)\mathrm{d}\zeta\mathrm{d}\theta\mathrm{d}\varphi = r^2 r_\zeta \sin\theta\mathrm{d}\zeta\mathrm{d}\theta\mathrm{d}\varphi \tag{6.70}$$

- Differential operators

  – Gradient

$$\nabla\phi = \phi_{,i}\mathbf{E}^i = \frac{\partial\phi}{\partial\zeta}\mathbf{E}^\zeta + \frac{\partial\phi}{\partial\theta}\mathbf{E}^\theta + \frac{\partial\phi}{\partial\varphi}\mathbf{E}^\varphi \tag{6.71}$$

  – Divergence

$$\begin{aligned}\nabla \cdot \mathbf{v} \quad &= V^i{}_{;i} = \frac{\partial V^i}{\partial x^i} + \frac{\partial \log\sqrt{|g|}}{\partial x^k} V^k = \\ &= \frac{\partial V^\zeta}{\partial\zeta} + \left(\frac{2r_\zeta}{r} + \frac{r_{\zeta\zeta}}{r_\zeta}\right) V^\zeta + \frac{\partial V^\theta}{\partial\theta} + \left(\frac{2r_\theta}{r} + \frac{\cos\theta}{\sin\theta} + \frac{r_{\zeta\theta}}{r_\zeta}\right) V^\theta + \frac{\partial V^\varphi}{\partial\varphi}\end{aligned} \tag{6.72}$$

  – Laplacian

$$\begin{aligned}\Delta\phi \quad &= \mathrm{div}(\nabla\phi) = (g^{ij}\phi_{,j})_{;i} = \frac{1}{\sqrt{|g|}}\frac{\partial}{\partial x^i}\left(\sqrt{|g|}g^{ij}\frac{\partial\phi}{\partial x^j}\right) = \\ &= g^{\zeta\zeta}\frac{\partial^2\phi}{\partial\zeta^2} + 2g^{\zeta\theta}\frac{\partial^2\phi}{\partial\zeta\partial\theta} + \frac{1}{r^2}\frac{\partial^2\phi}{\partial\theta^2} + \frac{1}{r^2\sin^2\theta}\frac{\partial^2\phi}{\partial\varphi^2} + \\ &\quad + \left[\frac{2}{rr_\zeta} - \frac{r_{\theta\theta}}{r^2 r_\zeta} - g^{\zeta\zeta}\frac{r_{\zeta\zeta}}{r_\zeta} - g^{\zeta\theta}\left(\frac{2r_{\zeta\theta}}{r_\zeta} - \frac{\cos\theta}{\sin\theta}\right)\right]\frac{\partial\phi}{\partial\zeta} + \frac{\cos\theta}{r^2\sin\theta}\frac{\partial\phi}{\partial\theta}\end{aligned} \tag{6.73}$$

  – Curl

$$\begin{aligned}\nabla \times \mathbf{v} \quad &= \epsilon^{ijk} V_{k;j}\mathbf{E}_i = \\ &= \frac{1}{r^2 r_\zeta \sin\theta}\left[\left(\frac{\partial V_\varphi}{\partial\theta} - \frac{\partial V_\theta}{\partial\varphi}\right)\mathbf{E}_\zeta + \left(\frac{\partial V_\zeta}{\partial\varphi} - \frac{\partial V_\varphi}{\partial\zeta}\right)\mathbf{E}_\theta + \left(\frac{\partial V_\theta}{\partial\zeta} - \frac{\partial V_\zeta}{\partial\theta}\right)\mathbf{E}_\varphi\right]\end{aligned} \tag{6.74}$$

– Material derivative

$$(\mathbf{a}\cdot\nabla)\mathbf{b} = A^j B^i{}_{;j}\mathbf{E}_i =$$

$$= \left[A^\zeta \frac{\partial B^\zeta}{\partial\zeta} + A^\theta \frac{\partial B^\zeta}{\partial\theta} + A^\varphi \frac{\partial B^\zeta}{\partial\varphi} + \frac{r_{\zeta\zeta}}{r_\zeta} A^\zeta B^\zeta + \left(\frac{r_{\zeta\theta}}{r_\zeta} - \frac{r_\theta}{r}\right)\left(A^\zeta B^\theta + A^\theta B^\zeta\right) + \right.$$
$$\left. + \frac{1}{r_\zeta}\left(r_{\theta\theta} - \frac{2r_\theta^2}{r} - r\right) A^\theta B^\theta + \frac{\sin\theta}{r_\zeta}\left(r_\theta\cos\theta - r\sin\theta\right) A^\varphi B^\varphi\right]\mathbf{E}_\zeta +$$
$$+ \left[A^\zeta \frac{\partial B^\theta}{\partial\zeta} + A^\theta \frac{\partial B^\theta}{\partial\theta} + A^\varphi \frac{\partial B^\theta}{\partial\varphi} + \frac{r_\zeta}{r}\left(A^\zeta B^\theta + A^\theta B^\zeta\right) + \frac{2r_\theta}{r} A^\theta B^\theta - \sin\theta\cos\theta A^\varphi B^\varphi\right]\mathbf{E}_\theta +$$
$$+ \left[A^\zeta \frac{\partial B^\varphi}{\partial\zeta} + A^\theta \frac{\partial B^\varphi}{\partial\theta} + A^\varphi \frac{\partial B^\varphi}{\partial\varphi} + \frac{r_\zeta}{r}\left(A^\zeta B^\varphi + A^\varphi B^\zeta\right) + \left(\frac{r_\theta}{r} + \frac{\cos\theta}{\sin\theta}\right)\left(A^\theta B^\varphi + A^\varphi B^\theta\right)\right]\mathbf{E}_\varphi \tag{6.75}$$

- Christoffel symbols (different from 0)

$$\begin{array}{lll}
\Gamma^\zeta_{\zeta\zeta} = \dfrac{r_{\zeta\zeta}}{r_\zeta} & \Gamma^\zeta_{\zeta\theta} = \Gamma^\zeta_{\theta\zeta} = \dfrac{r_{\zeta\theta}}{r_\zeta} - \dfrac{r_\theta}{r} & \Gamma^\zeta_{\theta\theta} = \dfrac{1}{r_\zeta}\left(r_{\theta\theta} - \dfrac{2r_\theta^2}{r} - r\right) \\
\Gamma^\zeta_{\varphi\varphi} = \dfrac{\sin\theta}{r_\zeta}(r_\theta\cos\theta - r\sin\theta) & \Gamma^\theta_{\zeta\theta} = \Gamma^\theta_{\theta\zeta} = \dfrac{r_\zeta}{r} & \Gamma^\theta_{\theta\theta} = \dfrac{2r_\theta}{r} \\
\Gamma^\theta_{\varphi\varphi} = -\sin\theta\cos\theta & \Gamma^\varphi_{\zeta\varphi} = \Gamma^\varphi_{\varphi\zeta} = \dfrac{r_\zeta}{r} & \Gamma^\varphi_{\theta\varphi} = \Gamma^\varphi_{\varphi\theta} = \dfrac{r_\theta}{r} + \dfrac{\cos\theta}{\sin\theta}
\end{array} \tag{6.76}$$

6.1.3. Multidomain and continuity conditions

When using multidomain we will deal with the problem of writing continuity conditions for the different variables on the boundaries between contiguous subdomains. The main issue is that the mapping presented in 6.1.1 have discontinuities in some of its derivatives, so we should take care of using the correct expressions for this conditions.

	Continuous between subdomains	
	Bonazzola	Linear mapping
r	Yes	Yes
r_ζ	No (Yes for fixed a_i)	No
$r_{\zeta\zeta}$	No	Yes
r_θ	Yes	Yes
$r_{\theta\theta}$	Yes	Yes
$r_{\zeta\theta}$	Yes	No

In the case of a scalar field $\phi(r,\theta)$, if ϕ is continuous between subdomains, the condition is easy

$$\phi^{(+)} = \phi^{(-)} \tag{6.77}$$

where $(+)$ and $(-)$ represent each side of the boundary. If we want ϕ to be derivable across the boundary, we would write a condition for the normal derivative $\hat{\mathbf{n}}\cdot\nabla\phi$ and not for $\frac{\partial\phi}{\partial\zeta}$,

$$\hat{\mathbf{n}}\cdot\nabla^{(+)}\phi^{(+)} = \hat{\mathbf{n}}\cdot\nabla^{(+)}\phi^{(+)} \tag{6.78}$$

where

$$
\begin{aligned}
\hat{\mathbf{n}} \cdot \nabla\phi &= \frac{\mathbf{E}^\zeta}{\sqrt{g^{\zeta\zeta}}} \cdot \left(\frac{\partial\phi}{\partial\zeta}\mathbf{E}^\zeta + \frac{\partial\phi}{\partial\theta}\mathbf{E}^\theta + \frac{\partial\phi}{\partial\varphi}\mathbf{E}^\varphi \right) \\
&= \sqrt{g^{\zeta\zeta}}\frac{\partial\phi}{\partial\zeta} + \frac{g^{\zeta\theta}}{\sqrt{g^{\zeta\zeta}}}\frac{\partial\phi}{\partial\theta} \\
&= \sqrt{1+\frac{r_\theta^2}{r^2}} \left(\frac{1}{r_\zeta}\frac{\partial\phi}{\partial\zeta} - \frac{r_\theta}{r^2+r_\theta^2}\frac{\partial\phi}{\partial\theta} \right)
\end{aligned}
\tag{6.79}
$$

We know that r and r_θ are continuous across the boundary and if ϕ is continuous then so is $\frac{\partial\phi}{\partial\theta}$, then the condition (6.78) becomes

$$
\frac{1}{r_\zeta^{(+)}}\left(\frac{\partial\phi}{\partial\zeta}\right)^{(+)} = \frac{1}{r_\zeta^{(-)}}\left(\frac{\partial\phi}{\partial\zeta}\right)^{(-)}
\tag{6.80}
$$

that is equivalent to saying $\left(\frac{\partial\phi}{\partial r}\right)^{(+)} = \left(\frac{\partial\phi}{\partial r}\right)^{(-)}$.

In the case of a vector field, the conditions are

$$
\begin{aligned}
r_\zeta^{(+)} V^{\zeta(+)} &= r_\zeta^{(-)} V^{\zeta(-)} \\
V^{\theta(+)} &= V^{\theta(-)} \\
V^{\varphi(+)} &= V^{\varphi(-)}
\end{aligned}
\tag{6.81}
$$

for continuity and

$$
\begin{aligned}
\left(\frac{\partial V^\zeta}{\partial\zeta}\right)^{(+)} + \frac{1}{r_\zeta^{(+)}}\left(r_{\zeta\zeta}^{(+)} V^{\zeta(+)} + r_{\zeta\theta}^{(+)} V^{\theta(+)} \right) &= \left(\frac{\partial V^\zeta}{\partial\zeta}\right)^{(-)} + \frac{1}{r_\zeta^{(-)}}\left(r_{\zeta\zeta}^{(-)} V^{\zeta(-)} + r_{\zeta\theta}^{(-)} V^{\theta(-)} \right) \\
\frac{1}{r_\zeta^{(+)}}\left(\frac{\partial V^\theta}{\partial\zeta}\right)^{(+)} &= \frac{1}{r_\zeta^{(-)}}\left(\frac{\partial V^\theta}{\partial\zeta}\right)^{(-)} \\
\frac{1}{r_\zeta^{(+)}}\left(\frac{\partial V^\varphi}{\partial\zeta}\right)^{(+)} &= \frac{1}{r_\zeta^{(-)}}\left(\frac{\partial V^\varphi}{\partial\zeta}\right)^{(-)}
\end{aligned}
\tag{6.82}
$$

for derivability. These conditions are based on the fact that the spherical components of the vector field are continuous and derivable scalar functions. For a physical boundary, it could happen that some components of the vector field are continuous (or derivable) and others are not, then none of the above conditions is correct.

6.2. Coordinate mapping in ESTER

The class `mapping` combines a `diff_gl` object with a `diff_leg` object to perform calculations in 2D. The prototype of this class is defined in `mapping.h`.

Let's see an example of initialization of a mapping object

```
mapping map;

map.set_ndomains(3);
map.set_npts(20);
map.set_nt(25);
map.set_nex(20);
map.init();

map.R.setrow(1,0.5*ones(1,map.nt));
map.R.setrow(2,1+0.1*sin(map.th)*sin(map.th));
map.R.setrow(3,2+0.3*sin(map.th)*sin(map.th));

map.remap();
```

First, we have declared the object

```
mapping map;
```

Then we set the number of domains

```
map.set_ndomains(3);
```

Our mapping will have 3 domains plus one external domain. We choose the number of points in the internal domains

```
map.set_npts(20);
```

That is, the 3 domains will have each one 20 points. We can also set a different number of points for each domain, for that we should do

```
int npts[3];
npts[0]=10;npts[1]=20;npts[2]=30;
map.set_npts(npts);
```

After that, we set the number of points in latitude (25) and in the external domain (20)

```
map.set_nt(25);
map.set_nex(20);
```

If we are not using the external domain, there is no need to change its number of points (It is 10 by default). There is one optional parameter that we can set before initialize the mapping

```
map.mode=MAP_BONAZZOLA;
```

to use the mapping with constant r_ζ at the boundaries presented in the previous section (this is the default), or

```
map.mode=MAP_LINEAR;
```

to use a linear mapping. Now we can proceed with the initialization

```
map.init();
```

If we change the resolution after this point, we should call `init()` again. Now we have a working object, with 3 spherical domains distributed uniformly from $r = 0$ to $r = 1$. If this setup is fine for us, then we are done with the initialization, but usually we will want to change to boundaries between domains. This is done using the matrix `R` with size $(n_{\text{domains}} + 1) \times n_\theta$. Each row of `R` defines a different boundary, starting with the row 0 that corresponds to the inner boundary. In the previous example, we have not changed the inner boundary, so our mapping contains the center of coordinates, but if we want the mapping to start, for example, at $r = 0.3$, we could just add

```
map.R.setrow(0,0.3*ones(1,map.nt));
```

In the definition of the boundaries we have used `map.th` that contains the values of the θ coordinate. Once we have set the boundaries we just call `remap()`

```
map.remap();
```

We can call `remap()` as many times as we want if we need to change the boundaries again.

After the initialization, a `mapping` object contains the following variables

Name	Type	Size	Description
`ndomains`	`int`		Number of domains
`nr`	`int`		Number of radial points
`nt`	`int`		Number of latitudinal points
`nex`	`int`		Number of radial points in the external domain
`npts`	`int[ndomains]`		Number of points in each domain
`R`	`matrix`	(ndomains+1,nt)	Domain boundaries
`eta`	`matrix`	(ndomains+1,nt)	$R(\theta = 0)$
`z`	`matrix`	(nr,1)	Spheroidal radial coordinate ζ
`th`	`matrix`	(1,nt)	Colatitude θ
`r`	`matrix`	(nr,nt)	Spherical radial coordinate r
`ex.r`	`matrix`	(nex,nt)	Spherical radial coordinate r in the external domain
`rz`, `rt`, `rzz`, `rtt`, `rzt`	`matrix`	(nr,nt)	r_ζ, r_θ, $r_{\zeta\zeta}$, $r_{\theta\theta}$, $r_{\zeta\theta}$
`ex.rz`, `ex.rt`, `ex.rzz`, `ex.rtt`, `ex.rzt`	`matrix`	(nex,nt)	r_ζ, r_θ, $r_{\zeta\zeta}$, $r_{\theta\theta}$, $r_{\zeta\theta}$ in the external domain
`gzz`, `gzt`, `gtt`	`matrix`	(nr,nt)	Elements of the metric tensor $g^{\zeta\zeta}$, $g^{\zeta\theta}$ and $g^{\theta\theta}$
`ex.gzz`, `ex.gzt`, `ex.gtt`	`matrix`	(nex,nt)	Elements of the metric tensor $g^{\zeta\zeta}$, $g^{\zeta\theta}$ and $g^{\theta\theta}$ in the external domain

`D`	`matrix_block_diag`	(nr,nr)	Differentiation matrix $\frac{\partial}{\partial\zeta}$
`ex.D`	`matrix`	(nex,nex)	Differentiation matrix $\frac{\partial}{\partial\zeta}$ in the external domain
`Dt`, `Dt_11`, `Dt_01`, `Dt_10`	`matrix`	(nt,nt)	Differentiation matrix $\frac{\partial}{\partial\theta}$ for each type of symmetry
`Dt2`, `Dt2_11`, `Dt2_01`, `Dt2_10`	`matrix`	(nt,nt)	Second order differentiation matrix $\frac{\partial^2}{\partial\theta^2}$ for each type of symmetry
`I`	`matrix`	(1,nr)	Integration matrix $\int_{\eta_0}^{\eta_{\mathrm{ndom.}}} \mathrm{d}\zeta$
`It`	`matrix`	(nt,1)	Integration matrix $\int_0^\pi \sin\theta \mathrm{d}\theta$ for symmetric functions
`J`	`matrix[4]`	(nr,nt)	Jacobian of the mapping
`ex.J`	`matrix[4]`	(nex,nt)	Jacobian of the mapping in the external domain
`gl`	`diff_gl`		Numerical differentiation object for the radial direction
`ex.gl`	`diff_gl`		Numerical differentiation object for the radial direction in the external domain
`leg`	`diff_leg`		Numerical differentiation object for the latitudinal direction

A given scalar field $\phi(\zeta,\theta)$ will be represented by a 2D matrix with size `nr`×`nt`, each element being the value of the function at each collocation point $\phi_{ij} = \phi(\zeta_i, \theta_j)$. With this representation, the operators acting on the radial direction are implemented using left multiplication while those acting in the latitudinal direction use right multiplication. Let's see some examples

$$\begin{aligned}
\frac{\partial\phi}{\partial\zeta} \quad &: \quad \texttt{(map.D,phi)} \\
\int_{\eta_0}^{\eta_{\mathrm{ndom.}}} \phi \,\mathrm{d}\zeta \quad &: \quad \texttt{(map.I,phi)} \\
\frac{\partial\phi}{\partial\theta} \quad &: \quad \texttt{(phi,map.Dt)} \\
\int_0^\pi \phi \sin\theta \,\mathrm{d}\theta \quad &: \quad \texttt{(phi,map.It)}
\end{aligned}$$

We can also use operators in both sides at the same time

$$\begin{aligned}
\frac{\partial^2\phi}{\partial\zeta\partial\theta} \quad &: \quad \texttt{(map.D,phi,map.Dt)} \\
\int_{\eta_0}^{\eta_{\mathrm{ndom.}}} \int_0^\pi \phi r^2 r_\zeta \sin\theta \,\mathrm{d}\zeta\mathrm{d}\theta \quad &: \quad \texttt{(map.I,phi*map.r*map.r*map.rz,map.It)}
\end{aligned}$$

Most of the members of a `mapping` object are in fact references to the corresponding member of the `diff_gl` or `diff_leg` object presented in the previous chapter, for example `map.D` is equivalent to `map.gl.D`.

We can use the interpolation functions defined in `diff_gl` and `diff_leg` for interpolation in ζ and θ respectively. The `mapping` class provides also a function for interpolating at some points (r_{ij},θ_{ij}) given in spherical coordinates

```
map.eval(phi,ri,thi,parity)
```

where `ri` and `thi` are matrices containing the interpolation points and parity is an integer representing the type of symmetry of `phi` (00, 01, 10 or 11). For symmetric functions (00), this parameter can be omitted.

6.2.1. Example

Let's take the example in section 5.3.1 and rewrite it for spheroidal coordinates, now using the `mapping` class.

```
/* The following example illustrates the use of the mapping
library in spheroidal coordinates */

#include<stdio.h>
#include"mapping.h"
#include"constants.h" //For the defintion of PI

//Function prototypes
double laplacian(matrix y,double r0,double th0);
double integral(matrix y);

// Define mapping objects as global variable
mapping map;
// Create references for spherical coordinates
matrix &r=map.r,&th=map.th;

int main() {

    //Initialize map. In the example we will use 2 domains
    map.set_ndomains(2);
    map.set_npts(100);
    map.set_nt(50);
    //map.set_nex(20); // We won't use the external domain
    map.init();

    map.R.setrow(0,1e-3*ones(1,map.nt));
                            // Use 1e-3 as the interior limit (instead of 0)
                            // to avoid a division by zero in the
                            // calculation of the laplacian
    map.R.setrow(1,0.5+0.1*sin(th)*sin(th));
    map.R.setrow(2,ones(1,map.nt));
    map.remap();

    matrix y;
```

```
    //Define the function y
    y=r*r*r*(1+sin(th)*sin(th));

    double lap_y,int_y;
    double r0=0.3,th0=PI/3;

    lap_y=laplacian(y,r0,th0);
    int_y=integral(y);

    printf("The value of the laplacian at (%f,%f) is %e \n ",r0,th0,lap_y);
    printf("The volume integral is %e \n ",int_y);

    return 0;

}

// Function for calculating the laplacian of y at (r0,th0)
double laplacian(matrix y,double r0,double th0) {

    matrix lap_y;

    matrix &gzz=map.gzz,&gzt=map.gzt,&gtt=map.gtt;
    matrix &rz=map.rz,&rzz=map.rzz,&rzt=map.rzt,&rt=map.rt,&rtt=map.rtt;
    matrix &Dt=map.Dt,&Dt2=map.Dt2;
    matrix_block_diag &D=map.D;

    lap_y=gzz*(D,D,y)+2*gzt*(D,y,Dt)+(y,Dt2)/r/r
        +(2./r/rz-rtt/r/r/rz-gzz*rzz/rz-gzt*(2*rzt/rz-cos(th)/sin(th)))*(D,y)
        +cos(th)/r/r/sin(th)*(y,Dt);

    lap_y=map.eval(lap_y,r0*ones(1,1),th0*ones(1,1));

    return lap_y(0);

}
// Function for calculating the volume integral of y
double integral(matrix y) {

    return 2*PI*(map.I,r*r*map.rz*y,map.It)(0);

}
```

The output will be the same that in the previous example

```
The value of the laplacian at (0.300000,1.047198) is 6.150000e+00
The volume integral is 3.490659e+00
```

Bibliography

Bonazzola, S., Gourgoulhon, E., & Marck, J.-A. 1998. Numerical approach for high precision 3D relativistic star models. *Phys. Rev. D*, **58**(10), 104020.