ESTER User guide

December 10, 2012

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`.

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. After setting the appropriate values for the compilation, we must start by doing

```
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.

We can now build the main program by doing

```
ester/src$ make
```

To remove intermediate files we can also do

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

The main library is created in `ester/lib/libester.so`, the header files are in `ester/include` and the executables in `ester/bin`. We can add the latter directory to the system path, for the *bash* shell

```
$ export PATH="your_root_directory/ester/bin:$PATH"
```

or, to make this change permanent, include this line in your `.bashrc` file.

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

```
$ star1d
```

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

```
$ star2d -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

To use the ESTER library in a C++ program you should write at the beginning of your source file

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

To facilitate the process of compiling and linking against the library and all its dependencies, we provide an automatically generated script `ester/utils/ester_build` so, all you have to do is

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

2 Basic usage

There are three executables provided with the library:

- `star1d` For calculating the structure of a 1D non-rotating star
- `star2d` For calculating the structure of a 2D rotating star
- `gen_output` For generating a custom output file

2.1. Configuration files

The main configuration file for `star1d` and `star2d` is `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 exists 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 `/NULL`. 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. The bounding surfaces of the domains are located at 0,0.2,0.5,0.65,0.75,0.85,0.95,0.99,1. 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. `star1d` input parameters

The input parameters for `star1d` 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
```

- `xif`. The position of each subdomain as a comma-separated list. The list should contain the first and the last points of the entire domain (that should be 0 and 1), having a total of `ndomains`+1 values. If only one value is specified (γ), the positions are calculated using the formula

$$\texttt{xif(i)} = 1 - \left(1 - \frac{i}{\texttt{ndomains}}\right)^{\gamma} \qquad \gamma > 0$$

  where $\gamma = 1$ corresponds to equally-spaced subdomains, for $\gamma > 1$ they are more concentrated near the surface, and the opposite for $\gamma < 1$.

- `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

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

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

- `conv`. The number of subdomains within the convective core. If `conv=0` the model will be completely radiative.
- `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. At the moment, only `simple` is implemented.
- `atm`. Type of atmosphere. At the moment, only `simple` is implemented.

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.

At the moment, the code does not permit to change the number of domains and/or their position when using an input file.

2.4. `star2d` INPUT PARAMETERS

Note that the input of `star2d` can be a non-rotating 1D model calculated with `star1d`.

The program `star2d` admits the same parameters than `star1d` 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 $\Omega = 0.7\Omega_c$ we can do:

```
$ star1d -M 5 -opa opal -o model1d
$ star2d -i model1d -nth 24 -Omega_bk 0.7 -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

`$ star1d -M 2.5 -o model1d -conv 0`	(Deactivate core convection to improve convergence)
`$ star1d -i model1d -o model1d -conv 1`	(Re-activate core convection)
`$ star2d -i model1d -nth 24 -Omega_bk 0.7 -o model2d`	(Using an intermediate value for rotation)
`$ star2d -i model2d -nth 32 -Omega_bk 0.9 -o model2d`	(Calculating the final model)

Executing `star2d` with `maxit=0` can be used to interpolate a model without recalculating it.

`$ star2d -i model -npts` *npts_new* `-nth` *nth_new* `-o model_interp -maxit 0`

Pressing Ctrl-C at any time during the execution of `star2d` will terminate the program, giving the possibility of finishing the current iteration and write the result in the output file.

2.6. Generating custom output files

The output files generated by `star1d` and `star2d` contain just the minimal information necessary to reconstruct the code. However, sometimes a more detailed output is required. This can be done using the program `gen_output` included in the distribution. This program reads a template from the standard input and write the result in the standard output. A typical call would be

`$ gen_output` *model_file* `<` *template_file* `>` *output_file*

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

- Plain text are copied from the template to the output file. It cannot contain the reserved characters `$` and \.
- Line breaks are ignored. To insert a line break in the output file we have to insert a blank line in the template.
- Variables from the model are written in the form `${`*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 codes for the model variables in the template file

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	*	*
Rp	Polar radius	*	*
Re	Equatorial radius	*	*
L	Luminosity	*	*
M/M_SUN	Mass in solar units	*	*

Rp/R_SUN	Polar radius in solar units	*	*
Re/R_SUN	Polar radius in solar units	*	*
L/L_SUN	Luminosity in solar units	*	*
r	Radius	*	*
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 diffusivity (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		*

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.

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

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 behavior 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 ...
[...]
```

2.7. 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 `star2d`. The variables in the models are defined as *numpy* arrays. Here is a little example:

```
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

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

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

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:

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

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

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

where $\mathcal{P}_{il}^{-1} = P_l(x_i)$ 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 does 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 shows 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 functions 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); // integral= ∫_{xif[0]=0}^{xif[3]=10} y(x)dx

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

```
#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-12,0.2,1.); // Use 1e-12 as the interior limit (instead of 0)
                              //to avoid the central singularity
    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).