ESTER User guide

October 9, 2012

Contents

1 Getting started **2**
1.1 Prerequisites 2
1.2 Installation 2
1.3 Checking the installation 3
1.4 Using the library 3

2 Basic usage **4**
2.1 Configuration files 4
2.2 `star1d` input parameters 5
2.3 `star2d` input parameters 7
2.3.1 Some recipes 7
2.4 Generating custom output files 7
2.5 Python module 11

3 General structure of the code **12**

4 Matrix Algebra. The `matrix` library. **13**
4.1 Matrix creation and manipulation 13
4.2 File input/output 15
4.3 Operators 16
4.4 Block diagonal matrices 17
4.5 Function reference 18
4.5.1 A note about methods and functions 19
4.5.2 Matrix manipulation 20
4.5.3 File input/output 23
4.5.4 Special matrices 24
4.5.5 Matrix functions 25
4.5.6 Mathematical functions 27
4.5.7 Block diagonal matrices 30

1 Getting started

1.1. Prerequisites

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

- BLAS and LAPACK, for matrix algebra
- PGPLOT (CPGPLOT) for graphics output

The library PGPLOT is optative and can be deactivated in the `Makefile` (`make.inc`) setting the variable `USE_PGPLOT=0`.

1.2. Installation

Unpack the file of the distribution (normally `ester.tar.gz`) in the desired directory

```
root$ tar zxvf ester.tar.gz
```

Go to the directory `ester/src`

```
root$ cd ester/src
```

To clean out any previous installation we can do

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

First we have to compile the third-party libraries included in the distribution. This only have to be done the first time (or after a `distclean`). At the moment there is only one library that needs to be build, the package for interpolating opacity tables created by Günter Houdek. After checking the `Makefile` located at `ester/tables/houdek/v9` we do

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

Now, we are prepared to compile the main library. The main variables for the compilation should be defined in a file named `make.inc` . Two examples are included, `make.inc.icc` and `make.inc.gcc`, for the Intel compiler and the GNU compiler respectively (tested under Ubuntu 12.04). After setting the proper values for the compilation we execute `make all` or just

```
root/ester/src$ make
```

To remove intermediate files we can also do

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

The main library is created in `ester/lib/libester.so`, the include 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 include this line in your `.bashrc` file.

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 0.7
```

This calculates the structure of a star rotating at 70% 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. `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

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

  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).
  - `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.

2.3. star2d INPUT PARAMETERS

The program `star2d` admits the same parameters than `star1d` plus some extra specific options:

- `nth`. The number of grid points in the latitude.
- `nex`. Nuber 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.

The default values are written in the file `ester/config/2d_default.par`. Note that the input of `star2d` can be a non-rotating 1D model calculated with `star1d`.

2.3. 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 finish the current iteration and write the result in the output file.

2.4. 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 rd^2/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	Γ_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 $T_{\rm eff}(\theta)$	*	*
gsup	Effective gravity at the surface $g_{\rm eff}(\theta)$	*	*
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.

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

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 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 ...
[...]
```

2.5. 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:

```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
```

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.

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. Function 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()

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. 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. 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.

4.5. 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 0 on success, 1 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 0 on success, 1 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. 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. 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. 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. 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.