The Gerris Tutorial

Version 0.9.2

Stéphane Popinet
popinet@users.sf.net

March 31, 2006

Contents

1 Introduction

This tutorial is a step-by-step introduction to the different concepts necessary to use Gerris. It is specifically designed for a end-user and is not a technical description of the numerical techniques used within Gerris. If you are interested by that, you should consult the bibliography section on the Gerris web site[1].

Various versions of this tutorial are available:

- Printable formats: both compressed Postscript[2] and PDF[3].
- HTML: direct link[4] or compressed archive[5].

In this tutorial I will assume that you are familiar with the Unix shell (and that you are running some version of a Unix system). Some knowledge of C programming would also be very helpful if you intend to extend Gerris with your own objects.

1.1 Installing and compiling Gerris

Gerris is written in C and uses two other C libraries which you need to install on your system first: the Glib library[6] and the GTS library[7]. If you want to run the parallel version of Gerris, you will also need some implementation of the MPI (Message Passing Interface) library. If you are using a parallel machine, MPI is very likely to be already on your machine.

If you are using a Linux system the Glib library is most probably already installed on your system. To be sure try this:

```
% glib-config --version
```

or this

```
% pkg-config glib-2.0 --modversion
```

If you get a result, you have the Glib library on your system, otherwise you will need to install it following the instructions on the Glib web site.

[1] http://gfs.sf.net
[2] http://gfs.sf.net/tutorial/tutorial.ps.gz
[3] http://gfs.sf.net/tutorial/tutorial.pdf
[4] http://gfs.sf.net/tutorial/tutorial/tutorial1.html
[5] http://gfs.sf.net/tutorial/tutorial.tar.gz
[6] http://www.gtk.org
[7] http://gts.sf.net

1.1.1 Using Debian packages

If you are using one of the Debian-based distribution (e.g. Debian, Knoppix, Ubuntu, etc...), you can install GTS and Gerris using your favourite Debian package management tool. Be aware however that the most recent version of Gerris may not be packaged yet for Debian (especially if you are using the "testing" or "stable" branches of Debian). To install Gerris using `apt-get` just type as root:

```
% apt-get install gerris
```

If you want to install only the Debian packages for GTS (and then compile Gerris from source) type:

```
% apt-get install libgts-dev
```

In all cases, this will also install the required version of Glib.

If you don't want to wait for an updated version of the official Debian package for Gerris, you can follow the development version of Gerris by using the unofficial Debian packages which are built every night automatically (see next section for more info). To do this just add the following lines to your `apt-get` repository list (usually `/etc/apt/sources.list`):

```
# GTS
deb http://gts.sourceforge.net/debian/ ./
# Gerris
deb http://gfs.sourceforge.net/debian/ ./
```

then do:

```
% apt-get update
% apt-get install libgts-snapshot gerris-snapshot gfsview-snapshot
```

this will install all the required libraries. When a new development version becomes available, doing an `apt-get upgrade` will automatically upgrade your versions of GTS, Gerris and GfsView.

1.1.2 Which sources?

When installing from source, you will need to decide which version of Gerris you want to install. Installing from the current official release (by following the *Download Gerris x.x.x* link on the Gerris web site) should be safe. If you don't want to wait for official releases, you can install more current versions of Gerris by following the *Gerris snapshot* link. These snapshots are generated automatically every night (when the code changes). They are more experimental than the official release but are tested both for correct compilation and correct execution by runnning the automated test suite[8]. If you choose to install from the snapshot, you will also need to install and regularly update GTS from the corresponding snapshots on the GTS web site.

1.1.3 Installing from source

If you choose to install everything from source, you will need to install the GNU Triangulated Surface Library (GTS). To do that, go on the GTS web site and download a recent source file package. Move it to your preferred location for unpacking source files and type:

```
% gunzip gts.tar.gz
% tar xvf gts.tar
% cd gts
```

You now need to decide where you want to install the library, if you have access to the root account, you can simply type:

[8] http://gfs.sf.net/tests/tests/index.html

```
% ./configure
% make
% su
% make install
% exit
```

which will install everything in `/usr/local` by default.

If you don't have access to the root password or want to install the library somewhere else like `/home/joe/local` for example, you can type instead:

```
% ./configure --prefix=/home/joe/local
% make
% make install
```

You can also type

```
% ./configure --help
```

to get a summary of the options taken by configure.

You probably want to add `/home/joe/local/bin` to your `PATH`, `/home/joe/local/lib` to your `LD_LIBRARY_PATH` and `/home/joe/local/lib/pkgconfig` to your `PKG_CONFIG_PATH`.

The next step is installing Gerris itself, do the same, go to the Gerris web site[9] download a recent source file package and type:

```
% gunzip gerris.tar.gz
% tar xvf gerris.tar
% cd gerris
% ./configure --prefix=/home/joe/local
% make
% make install
```

Both the 2D and 3D versions of the code are built. In this tutorial, we will for convenience only work with 2D simulations. However, all we will learn can be applied directly to 3D simulations.

We are now ready to start. Just to check that everything is okay try:

```
% gerris2D -V
```

1.1.4 Installing from the version control system

Installing from the version control system is the preferred way of installing Gerris if you want to change the code and submit changes for inclusion in the main Gerris distribution. I currently use darcs[10], a very nice distributed version control system. I will just give a recipe on how to install Gerris from the central darcs repository. You should consult the darcs manual[11] if you want to learn more about darcs.

After installing darcs[12], to get the current stable version of Gerris using darcs do:

```
% darcs get http://gfs.sf.net/darcs/gerris/gerris-stable
```

which will create a `gerris-stable` directory containing the source code. To build Gerris you will need the autoconf, automake toolsuite. The version of Automake must be >= 1.7 (it won't work with the 1.4 series). Then type:

```
% cd gerris-stable
% sh autogen.sh
```

[9] http://gfs.sf.net
[10] http://abridgegame.org/darcs/
[11] http://abridgegame.org/darcs/manual/
[12] http://darcs.net/DarcsWiki/CategoryBinaries

which will generate the configure script. Then follow the instructions of section 1.1.3.

The darcs repository is updated regularly with patches submitted by developers. To synchronise your current version with the most recent one type:

```
% cd gerris-stable
% darcs pull
% make
% make install
```

If you make changes to the source code and want to record these in darcs type:

```
% darcs record
```

You will also need to track the changes made to the GTS library if they are used by the current version of Gerris. I also use darcs to store GTS changes. To install GTS just repeat the instructions using `http://gts.sf.net/darcs/gts-mainline` as source repository.

Note that whenever you update/re-install GTS, you will need to rebuild Gerris. Just changing to the source directory of Gerris and typing `make` will rebuild Gerris if GTS has been reinstalled.

1.2 Simulation file

Gerris is a console-based program. It takes a *parameter* or *simulation* file as input and produces various types of files as output. Everything needed to run the simulation is specified in the parameter file, this includes:

- Layout of the simulation domain
- Initial conditions
- Boundary conditions
- Solid boundaries
- What to output (and when)
- Control parameters for the numerical schemes

2 A simple simulation file

In this section we will see how to write a simulation file for the *initial random vorticity* example in the Gerris web site gallery. First of all, it is always a good idea to run simulations in their own directory. Type this at your shell prompt:

```
% mkdir vorticity
% cd vorticity
```

As a starting point we will use the following simulation file: `vorticity.gfs`

```
1 2 GfsSimulation GfsBox GfsGEdge {} {
  GfsTime { end = 0 }
}
GfsBox {}
1 1 right
1 1 top
```

This is a valid simulation file but it does not do much as you can see by typing

```
% gerris2D vorticity.gfs
```

It is a good starting point however to explain the general structure of a simulation file.

2.1 A few comments on syntax

First of all, there are two types of parameters in a simulation file: *compulsory* and *optional* parameters. Optional parameters are always specified within a *braced* block (i.e. a block of text delimited by braces ({ `like this` }). They also often take the form

```
parameter = value
```

where `parameter` is an unique identifier (within this braced block). All the other parameters are compulsory parameters. For example, in `vorticity.gfs` both

```
  GfsTime { end = 0 }
```

and

```
end = 0
```

are optional parameters.

The second important syntax point regards the way various fields are delimited. Newline (or "carriage return") characters are generally used to delimitate different "objects" in the simulation file. The only case where this rule does not apply is within braced blocks defining optional arguments of the form

```
parameter = value
```

For example, in `vorticity.gfs` the following blocks of text are all objects:

- `1 2 GfsSimulation GfsBox GfsGEdge {} {`
  `GfsTime { end = 0 }`
  `}`
- `GfsTime { end = 0 }`
- `GfsBox {}`
- `1 1 right`
- `1 1 top`

Following this rule, `vorticty.gfs` could have been written equivalently as:

```
1 2 GfsSimulation GfsBox GfsGEdge {} { GfsTime {
  end = 0 }
}
GfsBox {}
1 1 right
1 1 top
```

2.2 Topology description of the simulation domain

Ok, so what are all these "objects" for? The first line of the simulation file defines a *graph* representing the general layout of the simulation domain and follows this syntax:

1st field number of nodes in the graph (`1`)

2nd field number of edges connecting the nodes (`2`)

3rd field object type for the graph (`GfsSimulation`)

4th field default object type for the nodes (`GfsBox`)

5th field object type for the edges (`GfsGEdge`)

6th field 1st optional parameters (braced block)

7th field 2nd optional parameters (braced block)

We then jump to the end of the 2nd optional parameters to line

```
GfsBox {}
```

which describes the first (and unique in this case) node of the graph. The first field is the object type of the node (`GfsBox`), the second field contains optional parameters. The following two lines

```
1 1 right
1 1 top
```

define the edges of the graph as follows:

1st field index of the starting node (`1`)

2nd field index of the ending node (`1`)

3rd field spatial direction in which the two nodes are connected (`right` and `top`)

The nodes are always indexed starting from one. The spatial directions are defined on figure 1. From this, we see that this file defines a simulation domain containing one node (a `GfsBox`) connected with itself in both the horizontal (`right`) and vertical (`top`) directions. By default, a `GfsBox` is a square

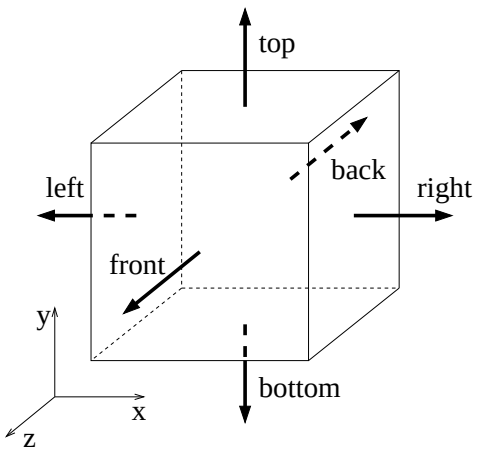

Figure 1: Definition of spatial directions

(in 2D) or a cube (in 3D) of size unity. The first node of the graph is always centered on the origin and is used as the reference to position the other nodes. We have consequently defined a square simulation domain of size unity, centered on the origin and using periodic boundary conditions.

2.3 Controlling the simulation

Now that we have defined the simulation domain and the boundary conditions, we need to specify the initial conditions, numerical schemes parameters and so on. This is all done within the second optional parameters block of the graph definition.

In our file we have for the moment only one object in this block:

```
GfsTime { end = 0 }
```

As its name indicate, this object defines the physical and the computational time. By "computational time" I mean the number of time steps performed. By default both the physical time and the time step number are zero when the program starts. It is possible to set different values using for example

```
GfsTime { t = 1.4 i = 32 end = 0 }
```

where `i` is the time step number and `t` is the physical time. The `end` identifier specifies that the simulation should stop when the physical time reaches the given value. It is also possible to stop the simulation when a specified number of time steps is reached, using the `iend` identifier. If both `end` and `iend` are specified, the simulation stops when either of these are reached. By default, both `end` and `iend` values are infinite.

Ok, let's then change this object to

```
GfsTime { end = 50 }
```

2.3.1 Spatial discretisation

The next step is to specify what spatial resolution we want for the discretisation. For the moment, the only thing we have defined is the root of the quad/octree. The whole domain is thus discretised with only one grid point...

We need to specify how we want to refine this initial root cell. This is done by using an `GfsRefine` object. We can do this by adding the line

```
GfsRefine 6
```

to the second optional parameter block. This is the simplest possible way to refine the initial root cell. We tell the program that we want to keep refining the cell tree (by dividing each cell in four children cells (in 2D, eight in 3D)) until the *level* of the cell is equal to five. The level of the root cell is zero, the level of all its children cells is one and so on recursively. After this refinement process completes we have created a regular Cartesian grid with $2^6 = 64$ cells in each dimension on the finest level (6).

2.3.2 Initial conditions

We now need to specify the initial conditions and the various actions (such as writing results, information messages etc...) we want to perform when the simulation is running. All these things are treated by Gerris as various types of *events*, all represented by objects derived from the same parent object `GfsEvent`.

Initial conditions are a particular type of event happening only once and before any other type of event, they are all derived from the same parent object `GfsInit`.

Gerris comes with a few different objects describing various initial conditions. As there is no way Gerris could provide all the different initial conditions users could think of, Gerris makes it easy for users to create their own initialisation objects by extending the `GfsInit` object class. In order not to have to recompile (or more exactly re-link) the whole code everytime a new class is added, Gerris uses dynamically linked *modules* which can be loaded at runtime. We will see later how to write your own modules.

For the moment, we will use the default `GfsInit` object. Just add the following lines to `vorticity.gfs`:

```
1 2 GfsSimulation GfsBox GfsGEdge {} {
  GfsTime { end = 50 }
  GfsRefine 6
  GfsInit {} {
    U = (0.5 - rand()/(double)RAND_MAX)
    V = (0.5 - rand()/(double)RAND_MAX)
  }
}
GfsBox {}
1 1 right
1 1 top
```

Using `GfsInit` it is possible to set the initial value of each of the simulation variables. By default all variables are set to zero initially. In our case we tell Gerris to define the two components of the velocity field `U` and `V` as C functions. The standard `rand` function of the C library returns a (pseudo)-random number between 0 and `RAND_MAX`. The two functions we defined thus set the components of the velocities in each cell as random numbers between -0.5 and 0.5.

This is a powerful feature of the parameter file. In most cases where Gerris requires a number (such as the `GfsRefine 6` line, a function of space and time can be used instead. For example, a valid parameter file could include:

```
...
  GfsRefine 6.*(1. - sqrt (x*x + y*y))
...
```

which would define a mesh refined in concentric circles.

Using this feature, it is possible to define most initial conditions directly in the parameter file.

2.3.3 Writing results

The `vorticity.gfs` file we have now is all Gerris needs to run the simulation. However, for this run to be any use, we need to specify how and when to output the results. This is done by using another class of objects: `GfsOutput`, derived from `GfsEvent`. Gerris comes with a number of these objects allowing to output various aspects of the simulation.

The general syntax for an `GfsEvent` object is as follows:

```
GfsEvent {
          start = 0.5 step = 1e-2 end = 3.4
          istart = 10 iend = 46
        }
```

this defines an event:

- starting whenever the physical time is larger than (or equal to) 0.5 or the time step number is larger than (or equal to) 10,
- ending whenever the physical time is strictly larger than 3.4 or the time step number is strictly larger than 46,
- and occurring every 10^{-2} physical time units.

It is also possible to specify an event step as a number of time steps using the `istep` identifier. Note, however, that you cannot specify both `step` and `istep` for the same event. By default, `start` and `istart` are zero and `end`, `iend`, `step` and `istep` are infinite.

An `GfsOutput` object is derived from `GfsEvent` and follows this syntax:

```
GfsOutput {} filename-%d-%f-%ld
```

The first part of the line `GfsOutput {}` defines the `GfsEvent` part of `GfsOutput` and follows the syntax above. In the remainder of this tutorial, I will use the following notation to express this inheritance of syntax:

```
[GfsEvent] filename-%d-%f-%ld
```

to avoid repeating the whole thing for every derived objects.

The second part `filename-%d-%f-%ld` specifies where to output things. The `%d`, `%f` and `%ld` characters are formatting strings which follow the C language syntax and will be replaced every time the event takes place according to:

`%d` integer replaced with the current process number (used when running the parallel version of Gerris).

`%f` floating-point number replaced with the current physical time.

`%ld` (long) integer replaced with the current time step number.

Of course, you are free not to specify any of these, in which case the output will just be appended to the same file every time the event takes place. There are also two filenames which have a special meaning: `stdout` and `stderr`, for the standard output and standard error of the shell respectively.

We now add the following to our simulation file:

```
1 2 GfsSimulation GfsBox GfsGEdge {} {
  GfsTime { end = 50 }
  GfsRefine 6
  GfsInit {} {
    U = (0.5 - rand()/(double)RAND_MAX)
    V = (0.5 - rand()/(double)RAND_MAX)
  }
  GfsOutputTime            { istep = 10 } stdout
  GfsOutputProjectionStats { istep = 10 } stdout
}
GfsBox {}
1 1 right
1 1 top
```

The first line we added tells the program to output information about the time every 10 time steps on the standard output. The second line outputs statistics about the projection part of the algorithm.

You can now run the code like this:

```
% gerris2D vorticity.gfs
```

and you should get an output in your console looking like this (you can stop the program using `Ctrl-C`):

```
step:       0 t:      0.00000000 dt:  0.000000e+00
MAC projection        before     after       rate
    niter:    0
    residual.bias:    0.000e+00  0.000e+00
    residual.first:   0.000e+00  0.000e+00   0.0
    residual.second:  0.000e+00  0.000e+00   0.0
    residual.infty:   0.000e+00  0.000e+00   0.0
Approximate projection
    niter:    0
    residual.bias:    0.000e+00  0.000e+00
    residual.first:   1.050e-14  1.050e-14   0.0
    residual.second:  1.612e-14  1.612e-14   0.0
    residual.infty:   7.105e-14  7.105e-14   0.0
step:      10 t:      0.02190704 dt:  2.801016e-03
MAC projection        before     after       rate
    niter:    5
    residual.bias:   -3.053e-16  1.403e-16
    residual.first:   3.365e+01  2.949e-05  16.3
    residual.second:  4.274e+01  4.676e-05  15.6
    residual.infty:   1.954e+02  3.285e-04  14.3
Approximate projection
    niter:    5
    residual.bias:    9.714e-17  2.874e-16
    residual.first:   3.322e+01  2.548e-05  16.7
    residual.second:  4.250e+01  4.062e-05  16.0
    residual.infty:   1.880e+02  3.380e-04  14.1
```

```
step:      20 t:      0.05278371 dt:  3.531551e-03
MAC projection        before     after       rate
    niter:    5
...
```

The lines starting with `step:` are written by `GfsOutputTime`. They give the time step number, corresponding physical time and the time step used for the previous iteration.

The other lines are written by `GfsOutputProjectionStats` and give you an idea of the divergence errors and convergence rate of the two projection steps (MAC and approximate) performed during the previous iteration. The various norms of the residual of the solution of the Poisson equation are given before and after the projection step. The `rate` column gives the average amount by which the divergence is reduced by each iteration of the multigrid solver.

Well, numbers are great but what about some images? What we want to do, for example, is output some graphical representation of a given scalar field. In 2D, a simple way to do that is to create an image where each pixel is coloured according to the local value of the scalar. Gerris provides an object to do just that: `GfsOutputPPM` which will create a PPM (Portable PixMap) image. This object is derived from a more general class used to deal with scalar fields: `GfsOutputScalar` following this syntax:

```
[GfsOutput] { v = U min = -1 max = 2.5 }
```

where as before the square brackets express inheritance from the parent class. The `v` identifier specifies what scalar field we are dealing with, one of:

`U`, `V` **(and `W` in 3D)** : components of the velocity.

`P` : pressure.

`C` : passive tracer.

`Vorticity` : vorticity (norm of the vorticity vector in 3D).

`Velocity` : norm of the velocity.

The `min` and `max` values specify the minimum and maximum values this scalar can take. If they are not given, they are computed every time the event takes place.

We can now use this in our simulation file:

```
1 2 GfsSimulation GfsBox GfsGEdge {} {
  GfsTime { end = 50 }
  GfsRefine 6
  GfsInit {} {
    U = (0.5 - rand()/(double)RAND_MAX)
    V = (0.5 - rand()/(double)RAND_MAX)
  }
  GfsOutputTime            { istep = 10  } stdout
  GfsOutputProjectionStats { istep = 10  } stdout
  GfsOutputPPM             { step = 1 } vorticity-%4.1f.ppm { v = Vorticity }
}
GfsBox {}
1 1 right
1 1 top
```

The code will output every 1 time units a PPM image representing the vorticity field. The result will be written in files named: `vorticity-00.0.ppm`, `vorticity-01.0.ppm`... (if the `%4.1f` thing is not familiar, consult a C book or try `% man 3 printf`).

If you re-run the program using this new simulation file, you will get a number of PPM files (51 to be precise) you can then visualise with any image editing or viewing tool. I would recommend the

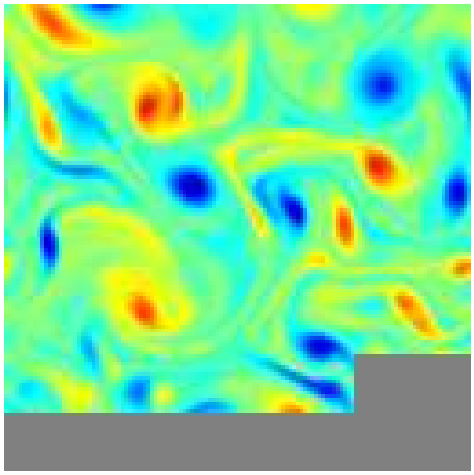

Figure 2: Vorticity field for the initial random vorticity problem at $t = 18$.

very good ImageMagick toolbox[13]. If you run a Linux box, these tools are very likely to be already installed on your system. Try typing this in your working directory:

```
% display *.ppm
```

If it works, you should see a small (64x64) image representing the initial vorticity field. If you click on it, a menu will appear. Select File→Next and look at the evolution of the vorticity field with time (you can also use the space bar and backspace key to change back and forth). You might also want to try the `animate *.ppm` command. Read the man pages of ImageMagick if you want to know more. Note that you can use these tools also while Gerris is running (and creating new images). With a bit of patience you will get the image on figure 2 at $t = 18$ (resolution has been increased to 128×128).

Before we carry on, we are going to make two modifications to the simulation file. First of all, it is not really handy to generate one file for every image generated. ImageMagick (and most other programs) can deal with multiple PPM images contained within the same file. Secondly, in the sequence of images we generate, a given value of the vorticity does not always correspond to the same colour (because the minimum and maximum values of the vorticity can vary in time). We can fix that like this:

```
1 2 GfsSimulation GfsBox GfsGEdge {} {
  GfsTime { end = 50 }
  GfsRefine 6
  GfsInit {} {
    U = (0.5 - rand()/(double)RAND_MAX)
    V = (0.5 - rand()/(double)RAND_MAX)
  }
  GfsOutputTime            { istep = 10   } stdout
  GfsOutputProjectionStats { istep = 10   } stdout
  GfsOutputScalarStats     { istep = 10   } stdout { v = Vorticity }
  GfsOutputPPM             { step = 0.1 } vorticity.ppm {
    v = Vorticity
    min = -10
    max =  10
  }
}
GfsBox {}
```

[13] http://www.imagemagick.org

```
1 1 right
1 1 top
```

We have now specified fixed bounds for the vorticity (using the `min` and `max` identifiers). Each PPM image will be appended to the same file: `vorticity.ppm`.

How did I choose the minimum and maximum values for the vorticity? The line `GfsOutputScalarStats { istep = 10 } stdout { v = Vorticity }`, writes the minimum, average, standard deviation and maximum values of the vorticity. By re-running the simulation and looking at these values it is easy to find a suitable range.

3 A more complex example with solid boundaries

In this section we will see how to set up a simulation for the flow past a solid body (a half-cylinder) in a narrow channel. While doing that we will also encounter new ways of displaying simulation results.

3.1 Domain geometry and boundary conditions

What we want is a narrow channel (4×1 for example). From the previous example, we know that we can build it like this:

```
4 3 GfsSimulation GfsBox GfsGEdge {} {
  GfsTime { end = 0 }
}
GfsBox {}
GfsBox {}
GfsBox {}
GfsBox {}
1 2 right
2 3 right
3 4 right
```

i.e. four boxes, box 1 connected to box 2 horizontally (to the right), box 2 connected to box 3 horizontally and box 3 connected to box 4 horizontally. Box 1 is centered on the origin and is of size one. All the other boxes are positioned accordingly. We now have our 4×1 rectangular domain.

3.1.1 Boundary conditions

What about boundary conditions? By default Gerris assumes that boundaries are solid walls with slip conditions for the velocity (i.e. the tangential stress on the wall is zero). For the moment we then have defined a rectangular box closed on all sides by solid walls.

What we really want is to specify an input velocity on the left side of the box and some sort of output condition on the right side. We can do that like this:

```
4 3 GfsSimulation GfsBox GfsGEdge {} {
  GfsTime { end = 0 }
}
GfsBox { left = GfsBoundaryInflowConstant 1 }
GfsBox {}
GfsBox {}
GfsBox { right = GfsBoundaryOutflow }
1 2 right
2 3 right
3 4 right
```

The whole left side of the first (leftmost) box is now defined to be a `GfsBoundaryInflowConstant` object and the whole right side of the last (rightmost) box a `GfsBoundaryOutflow` object. Again,

boundary conditions objects are all derived from the `GfsBoundary` object and, as initial conditions, new objects can be easily written by the user (see also section 4.1).

We see that `GfsBoundaryInflowConstant` takes one argument which is the value of the (constant) normal velocity applied to this boundary. All the other variables (pressure, tracer concentration etc...) follow a zero gradient condition.

`GfsBoundaryOutflow` implements a simple outflow boundary condition where the pressure is set to zero as well as the gradient of all other quantities.

3.1.2 Solid boundaries

We now have an empty "wind tunnel" with a constant inlet velocity of norm unity. Gerris can deal with arbitrarily complex solid boundaries embedded in the quad/octree mesh. The geometry of the solid boundaries is described using GTS triangulated surfaces. In 2D, using 3D triangulated surfaces seems overkill, as 2D curves would be enough. However, Gerris being both a 2D and 3D code it deals with 2D solid boundaries exactly as with 3D ones, even if the simulation is done only on a 2D cross-section.

Creating 3D polygonal surfaces is not an easy job and is clearly outside the scope of this tutorial. There are a number of utilities you can use to do that, including big commercial CAD packages. In general, once you have created a polygonal surface with one of these tools it should be relatively easy to convert it to the file format used by GTS. In particular, most CAD packages can export to the STL (stereolithography) format which is easily converted to the GTS file format using the `stl2gts` utility which comes with the library.

This tutorial comes (handily) with one such file: `half-cylinder.gts`[14]. You can visualise the surface it describes using a program called Geomview[15]. To do this, you first need to convert the GTS file to a format Geomview understands. This can be done using the `gts2oogl` utility like this:

```
% gts2oogl < half-cylinder.gts > half-cylinder.oogl
```

(OOGL is the file format used by Geomview). `gts2oogl` has a number of options. You can have a short explanation of what they do by typing:

```
% gts2oogl -h
```

If you now start geomview like this:

```
% geomview half-cylinder.oogl
```

and play around with the pan/rotate/zoom functions of Geomview (read the manual for details), you should see something like the image on figure 3. You can notice that this is a proper 3D object, even

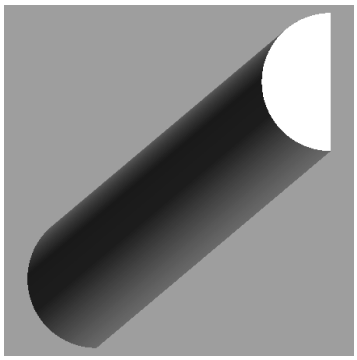

Figure 3: Geomview representation of `half-cylinder.gts`

if we are only going to simulate the flow in a 2D cross-section. It is also important that the object is

[14] http://gfs.sf.net/half-cylinder.gts
[15] http://www.geomview.org

"tall" enough so that it spans the entire "height" of the 2D domain, as if we were going to simulate the flow around it in a proper 3D channel with a square cross-section. The orientation of the surface is also important to define what is inside (the solid) and what is outside (the fluid).

We can now insert this object in the simulation domain like this:

```
4 3 GfsSimulation GfsBox GfsGEdge {} {
  GfsTime { end = 0 }
  GtsSurfaceFile half-cylinder.gts
}
GfsBox { left = GfsBoundaryInflowConstant 1 }
GfsBox {}
GfsBox {}
GfsBox { right = GfsBoundaryOutflow }
1 2 right
2 3 right
3 4 right
```

add what mesh refinement we want and a few things to output:

```
4 3 GfsSimulation GfsBox GfsGEdge {} {
  GfsTime { end = 9 }
  GfsRefine 6
  GtsSurfaceFile half-cylinder.gts
  GfsInit {} { U = 1 }
  GfsOutputBoundaries {} boundaries
  GfsOutputTime { step = 0.02 } stdout
  GfsOutputProjectionStats { step = 0.02 } stdout
  GfsOutputPPM { step = 0.02 } vorticity.ppm {
    min = -100 max = 100 v = Vorticity
  }
  GfsOutputTiming { start = end } stdout
}
GfsBox { left = GfsBoundaryInflowConstant 1 }
GfsBox {}
GfsBox {}
GfsBox { right = GfsBoundaryOutflow }
1 2 right
2 3 right
3 4 right
```

I have added a new `GfsOutput` object we haven't seen yet: `GfsOutputTiming`. This object writes a summary of the time taken by various parts of the solver. You might also have noticed the unusual `start = end` bit ; this just specifies that this event will only happen once at the end of the simulation.

Another new output object is `GfsOutputBoundaries`. This object writes a geometrical summary (in OOGL/Geomview format) of the mesh used, including boundary conditions, solid boundaries and so on.

We also initialise the velocity field on the whole domain to a constant value (1,0,0). We could have left the velocity field to its default value of (0,0,0) but, given that we impose inflow boundary conditions, it would have meant that the initial velocity would have been strongly divergent. Gerris always starts a simulation by a projection step (to fix problems like this) but it is always a good idea to start with the best possible velocity field.

We can now run the code:

```
% gerris2D half-cylinder.gfs
```

It is going to take a while to complete, but remember that you can look at files while they are being generated. The first file which will be generated is `boundaries`. If you load it in Geomview, you should

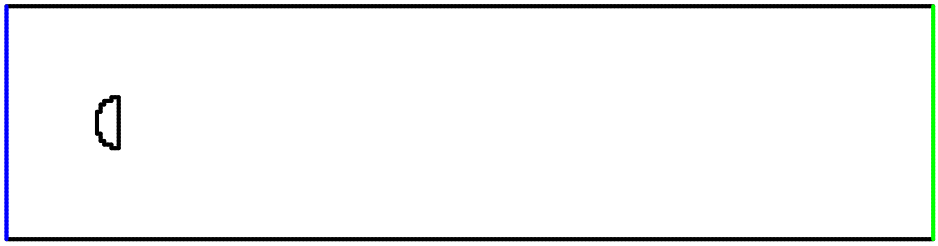

Figure 4: Representation of boundary conditions and solid boundaries

get something like figure 4 (you probably want to disable automatic normalization in Geomview by selecting Inspect→Appearance→Normalize→None). The black lines represent the boundaries between solid cells and fluid cells. If you zoom in on the half-cylinder, you will see that it is represented by lines following the grid (it is "lego-looking"). This does not mean that the "real" (i.e. computational) solid boundary is also lego-looking because fluid cells can be cut by the solid boundaries, in which case the algorithm properly takes into account the corresponding cell geometry.

Each `GfsBoundary` object is colour-coded. From the colours in the picture we see that we have indeed an inflow boundary condition on the left side (blue) and an outflow boundary condition on the right side (green).

You can also load in the full half-cylinder geometry we created before: `half-cylinder.oogl` or visualise the PPM files using `animate` and `display` as in the previous example. By the way, a useful feature of `display` is that you can zoom in by clicking on the middle button in the image being displayed.

3.2 Saving the whole simulation

Hmm, this simulation is taking quite a while... What if we want to stop the simulation, make some modifications to the simulation file and restart where we left from? Or equivalently, save the whole simulation at regular intervals for latter post-processing?

You can do this using the `GfsOutputSimulation` object. Like this for example:

```
GfsOutputSimulation { step = 0.1 } half-cylinder-%3.1f.gfs {
  variables = U,V,P
}
```

where `variables` defines which variables you want to save. By default all the variables are saved.

If you now re-run the simulation, you will get a new file every 0.1 time units. This file is a valid simulation file (like `half-cylinder.gfs`) and you can use it directly to restart the simulation from this point onward. If you edit it, you will see that the general structure is the same as usual but for five pretty big chunks of data.

The first chunk starts with `GtsSurface` and is just the data contained in `half-cylinder.gts` but this time embedded directly (by using `GtsSurface` rather than `GtsSurfaceFile`) into the simulation file. The goal there is to have fully self-contained simulations files which you can just move around without having to keep track of twenty different files.

The four other chunks are each associated with a `GfsBox` and contain both the topology of the corresponding cell tree but also the associated physical data, solid boundary definitions etc...

You can of course edit this file, add new outputs and so on and restart the simulation from where you left it.

3.3 Visualisation

Several tools can be used to visualise Gerris simulation files, either the standalone GfsView which interfaces directly with Gerris or external viewers such as geomview, openDX or MayaVi.

3.3.1 GfsView

GfsView is a tool written specifically to visualise Gerris simulation files. It is still young but fully usable both for 2D and 3D simulations. Its main advantage over other options and the reason for its existence is that it makes full use of the adaptive nature of the octree representation at the visualisation level. The octree structure is used within GfsView to dynamically select the appropriate level of refinement depending on the viewpoint, zoom and rendering speed. It is also used to efficiently compute complex geometrical entities such as isosurfaces or cut-planes.

The more classical viewers such as openDX or MayaVi are designed for either regular Cartesian grids or fully-unstructured meshes and do not take advantage of the octree representation (worse still, the octree representation first needs to be converted to Cartesian or fully-unstructured meshes before being imported into these programs).

To install GfsView, you need to have the Gtk+[16] toolkit installed on your system. If you are running a Linux machine, Gtk+ is most probably already installed. You will also need the GtkGlExt[17] OpenGL extension to Gtk+.

If you are running a Debian-based system, you can install these packages using

```
% apt-get install libgtkglext1-dev
```

If you then download a recent version of GfsView from the Gerris web site (either an official release or a snapshot) and do the now classical:

```
% gunzip gfsview.tar.gz
% tar xvf gfsview.tar
% cd gfsview
% ./configure --prefix=/home/joe/local
% make
% make install
```

you will be able to start GfsView using:

```
% gfsview2D half-cylinder-0.5.gfs
```

Note that you can also install the most recent GfsView version using darcs and `http://gfs.sourceforge.net/darcs/g` as source repository (you will also need to install Gerris this way, see section 1.1.4 for details).

Clicking on "Linear", "Vectors" and "Solid" in the toolbar and changing the vector length by editing the properties of the "Vectors" object (select the object then choose "Edit→Properties") you should be able to get something looking like figure 5. You can pan by dragging the right mouse button, zoom by dragging the middle button and rotate by dragging the left button.

While by no means complete, you can already do many things with GfsView. I hope it is fairly user-friendly so just play with it and discover for yourself.

3.3.2 Some post-processing using `gfs2oogl`

Gerris comes with a utility called `gfs2oogl` which converts simulation files to various representations in OOGL format. We are just going to look at two types of representations `gfs2oogl` can do: scalar field cross-sections and vector fields.

First of all, you can access a small summary of the options of `gfs2oogl` by typing:

```
% gfs2oogl2D -h
```

By default `gfs2oogl` will generate the same output as `GfsOutputBoundaries` like this:

```
% gfs2oogl2D < half-cylinder-0.1.gfs > boundaries.oogl
```

To generate an OOGL representation of a scalar field (a coloured square for each discretisation cell) do this:

[16] http://www.gtk.org
[17] http://gtkglext.sourceforge.net/

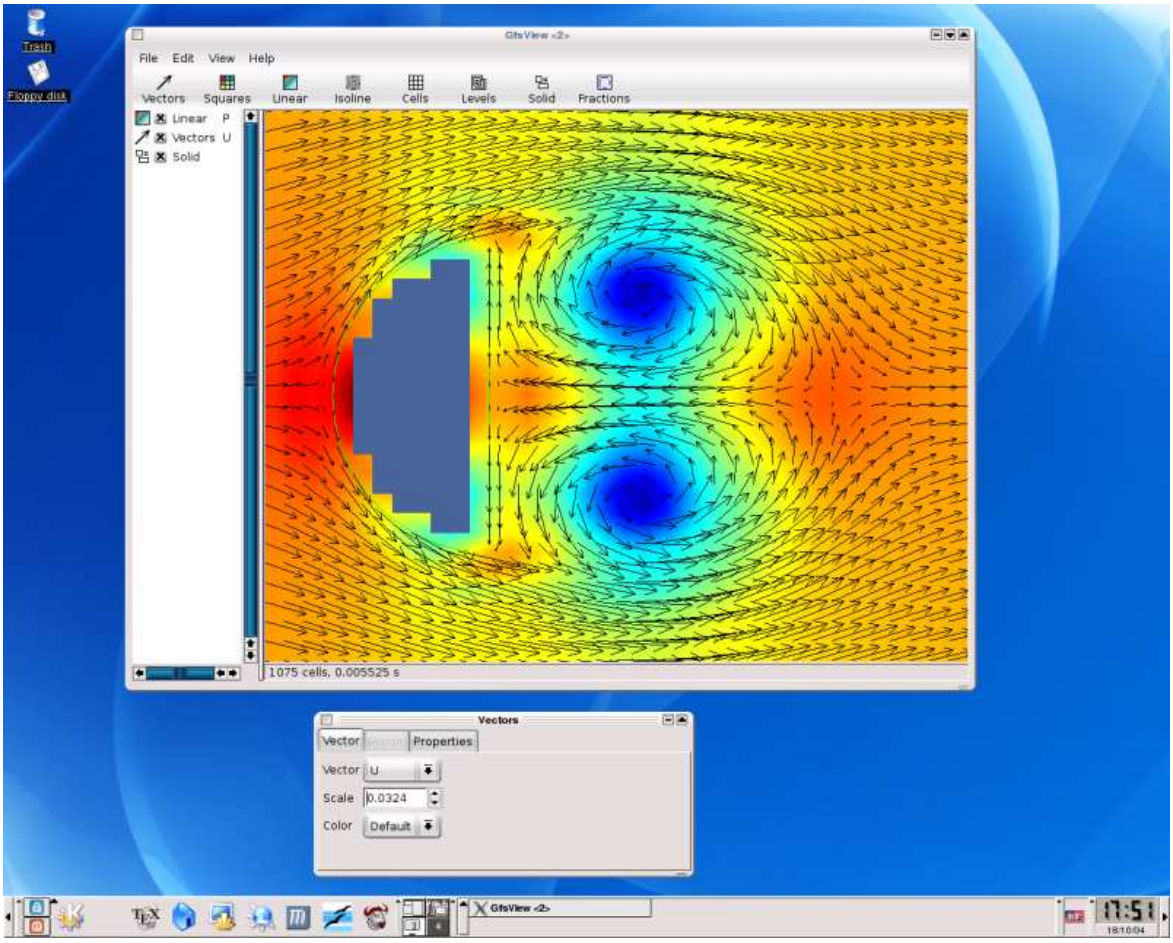

Figure 5: Screenshot of a GfsView session.

```
% gfs2oogl2D -S -z 0 -c Vorticity < half-cylinder-0.5.gfs > squares.oogl
```

which tells `gfs2oogl` to do a cross-section for $z = 0$ (`-z 0`) represented by squares (`-S`) and colored according to the local vorticity (`-c Vorticity`). To generate a vector field representing the velocity try:

```
% gfs2oogl2D -V 2 -z 0 < half-cylinder-0.5.gfs > vectors.oogl
```

where `-V 2` specifies that the maximum length of the vector is twice the dimension of the smallest cell in the domain.

If you now load all these files in Geomview and do a bit of panning and zooming around (and possibly tune things like face shading) you should get an image looking like figure 6.

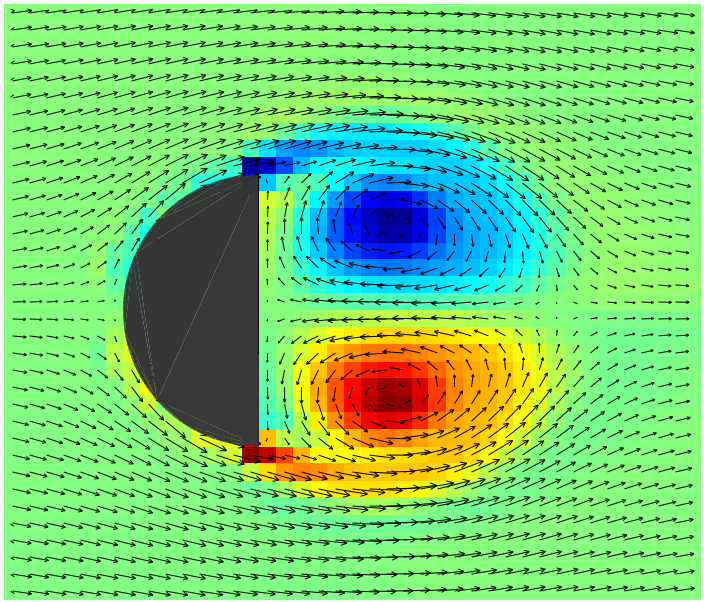

Figure 6: Scalar and vector representation generated using `gfs2oogl`.

3.3.3 Visualisation with OpenDX

If you have OpenDX[18] installed on your system, the installation process described above should have compiled and installed an OpenDX module which allows direct import of Gerris simulation files into an OpenDX visual program.

In order to be able to use this module you need to start OpenDX with a command line looking like:

```
% dx -mdf /home/joe/local/lib/gerris/dx2D.mdf
```

if you want to import a 2D simulation file or

```
% dx -mdf /home/joe/local/lib/gerris/dx3D.mdf
```

for a 3D simulation file (where `/home/joe/local` should be replaced with the proper installation directory). This tells OpenDX to load the module described by the given *Module Description File*.

If everything went correctly, within the OpenDX Visual Program Editor you should be able to select a tool named "ImportGfs2D" (resp. "ImportGfs3D") belonging to the "Import and Export" category. As its name suggests this tool reads a Gerris Flow Solver simulation file and produces a group containing named fields. A field called "solid" contains the geometry of the solid boundaries.

[18] http://www.opendx.org

The other fields represent the scalar or vector data contained in the simulation file. Vector fields are constructed from the velocity components (the resulting vector field is called "U" in OpenDX) and from any three components with names ending in "x", "y" or "z" and with a common root. This common root is used as the OpenDX field name. All the remaining fields are considered as scalar fields.

If you are not familiar with OpenDX, you should read the available documentation[19] and go through the integrated tutorial. You can use the visual program called `gfs2D.net`[20] provided with this tutorial as a starting point. Typing

```
% dx -mdf /home/joe/local/lib/gerris/dx2D.mdf -program gfs2D.net
```

in the directory containing your `half-cylinder-0.5.gfs` file (you also need to add the path to `gfs2D.net` if it is not contained in the same directory) and selecting the Execute→Execute Once menu should bring up windows looking like figure 7.

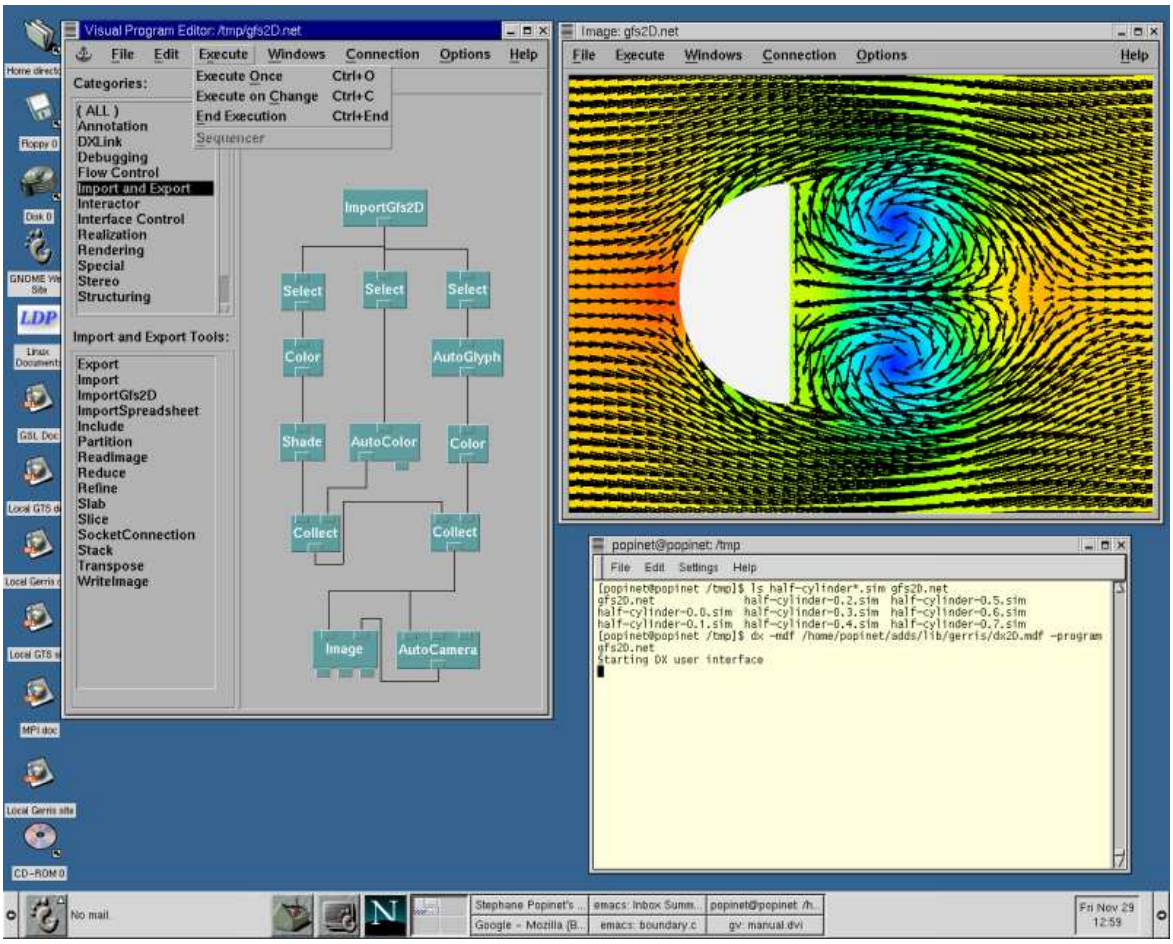

Figure 7: Screenshot of an OpenDX session using the ImportGfs2D module.

3.4 Using dynamic adaptive mesh refinement

For the moment our simulation is not very well resolved. We could always change the `GfsRefine 6` line to something bigger but it would not make really good use of the quadtree approach used in

[19] http://www.opendx.org/support.html#docs
[20] http://gfs.sf.net/gfs2D.net

Gerris. A code using a simple regular Cartesian grid approach would be faster and would produce the same results. Instead we are going to use *dynamic adaptive mesh refinement*, where the quadtree structure of the discretisation is used to adaptively follow the small structures of the flow, thus concentrating the computational effort on the area where it is most needed.

This is done using yet another object class: `GfsAdapt`, also derived from `GfsEvent`. Various criteria can be used to determine where refinement is needed. In practice, each criterium will be defined through a different object derived from `GfsAdapt`. If several `GfsAdapt` objects are specified in the same simulation file, refinement will occur whenever at least one of the criteria is verified.

For this first example, we will use a simple criterium based on the local value of the vorticity. A cell will be refined whenever

$$\frac{|\nabla \times \mathbf{v}|\Delta x}{\max |\mathbf{v}|} > \delta,$$

where Δx is the size of the cell and δ is a user-defined threshold which can be interpreted as the maximum angular deviation (caused by the local vorticity) of a particle traveling at speed $\max |\mathbf{v}|$ across the cell. This criterium is implemented by the `GfsAdaptVorticity` object.

The general syntax for an `GfsAdapt` object is:

```
[GfsEvent] { mincells = 1 maxcells = 100000 minlevel = 1 maxlevel = 10 cmax = 1e-2 }
```

where `mincells` specifies the minimum number of cells in the domain, `maxcells` the maximum number of cells, `minlevel` the level below which it is not possible to coarsen a cell, `maxlevel` the level above which it is not possible to refine a cell and `cmax` the maximum cell cost. The default values are 0 for `minlevel` and `mincells` and infinite for `maxlevel` and `maxcells`. An important point is that, for the moment, it is not possible to dynamically refine solid boundaries. A simple solution to this restriction is to always refine the solid boundary with the maximum resolution at the start of the simulation and to restrict the refinement using the `maxlevel` identifier in `GfsAdapt`.

What happens if the maximum number of cells is reached? The refinement algorithm will keep the number of cells fixed but will minimize the maximum cost over all the cells. This can be used for example to run a constant-size simulation where the cells are optimally distributed across the simulation domain. This would be done by setting `maxcells` to the desired number and `cmax` to zero.

Following this we can modify our simulation file:

```
4 3 GfsSimulation GfsBox GfsGEdge {} {
  GfsTime { end = 9 }
  GfsRefine 7
  GtsSurfaceFile half-cylinder.gts
  GfsInit {} { U = 1 }
#  GfsOutputBoundaries {} boundaries
  GfsAdaptVorticity { istep = 1 } { maxlevel = 7 cmax = 1e-2 }
  GfsOutputTime { step = 0.02 } stdout
  GfsOutputBalance { step = 0.02 } stdout
  GfsOutputProjectionStats { step = 0.02 } stdout
  GfsOutputPPM { step = 0.02 } vorticity.ppm {
    min = -100 max = 100 v = Vorticity
  }
  GfsOutputSimulation { step = 0.1 } half-cylinder-%3.1f.gfs {
    variables = U,V,P
  }
  GfsOutputTiming { start = end } stdout
}
GfsBox { left = GfsBoundaryInflowConstant 1 }
GfsBox {}
GfsBox {}
GfsBox { right = GfsBoundaryOutflow }
1 2 right
2 3 right
```

```
3 4 right
```

We have added two lines and commented out (using `#`) the line outputting the boundaries (we don't need that anymore, we have the simulation files).

The first line we added says that we want to refine dynamically the mesh through the `GfsAdaptVorticity` object applied every timestep (`istep = 1`). The δ parameter (`cmax`) is set to 10^{-2}.

The second line we added is a new `GfsOutput` object which displays the "balance" of the domain sizes across the different processes (when Gerris is ran in parallel). We will use this to monitor how the number of cells evolves with time as the simulation refines or coarsens the mesh according to our vorticity criterium.

We can now run this new simulation. If the previous simulation did not complete yet, don't be afraid to abort it (`Ctrl-C`), this one is going to be better (and faster).

```
% gerris2D half-cylinder.gfs
```

If we now look at the balance summary written by `GfsOutputBalance`, we see that initially (`step: 0`) the total number of cells (on all levels) is 86966, which corresponds to a constant resolution of $4 \times 2^7 \times 2^7 = 512 \times 128$. At step 10 the number of cells is down to 990 with a corresponding increase in computational speed. If we now look at the first simulation file we saved, using:

```
% gfs2oogl2D < half-cylinder-0.1.gfs > boundaries
% gfs2oogl2D -S -z 0 -c Vorticity < half-cylinder-0.1.gfs > squares.oogl
```

we obtain figure 8 showing not only the domain boundaries as usual, but also the boundaries (thin black lines) between different levels of refinement. We see that the mesh is very refined around the

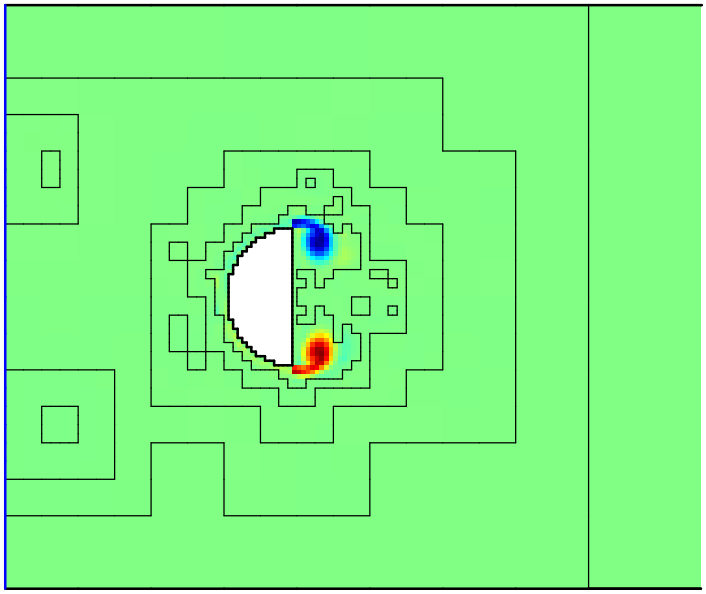

Figure 8: Dynamic adaptive mesh refinement $t = 0.1$

solid and around the two vortices developing at the trailing edge and very coarse (one cell per box only) on the downstream part of the domain. If you are not sure what these thin black lines represent, just switch on the edge representation in Geomview (using the Inspect→Appearance menu). You will get a picture looking like figure 9, showing all the cells used for the discretisation. As the simulation goes on, you can see the number of cells in the domain increase as the trailing vortices develop. With a bit of patience you will get to figure 10 showing the fully developed Von Karman vortex street with patches of increased resolution following each vortex. Even when the flow is fully developed using adaptive mesh refinement still saves a factor of ˜6 in time and memory use. The advantage of adaptive mesh refinement is even more obvious in situations where it is necessary to use very large domains to

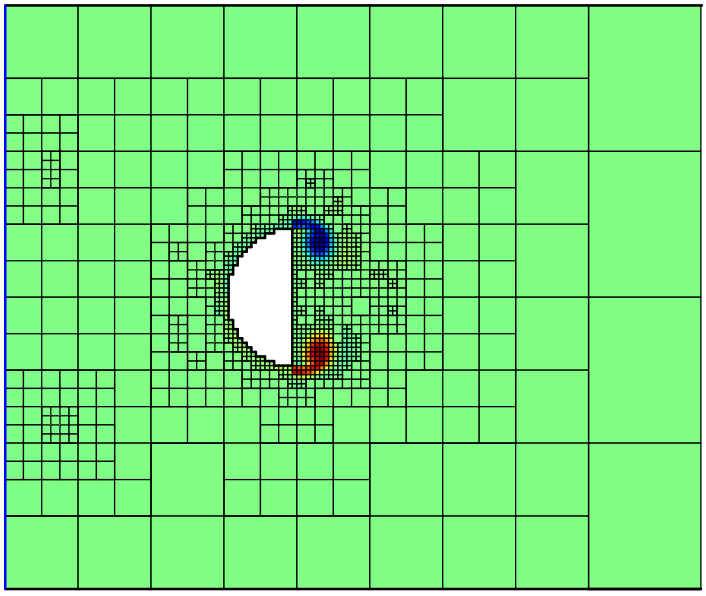

Figure 9: Dynamic adaptive mesh refinement $t = 0.1$. Detail of the cells.

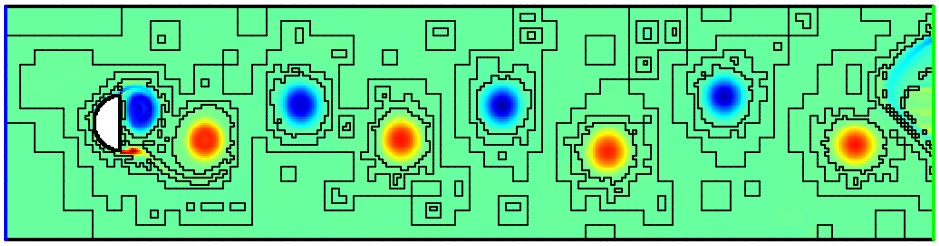

Figure 10: Dynamic adaptive mesh refinement $t = 9$.

avoid any contamination of the solution by the boundary conditions. You should also try to `animate vorticity.ppm` which by now should give you a nice animation of the developing trailing vortices becoming unstable and generating the Von Karman street. If ImageMagick is properly installed on your system you can also try:

```
% convert vorticity.ppm vorticity.mpg
```

which will produce a much smaller MPEG video file, suitable for distribution through the network.

4 Going further

4.1 More on boundary conditions

Up to now we have only dealt with "pre-packaged" boundary conditions such as `GfsBoundaryInflowConstant` and `GfsBoundaryOutflow`. What if you need more specific boundary conditions?

For most practical problems, boundary conditions can be reduced to two main categories: Dirichlet boundary conditions for which the value of the variable is set and Neumann boundary conditions for which the value of the derivative of the variable is set. As we have seen earlier, the default boundary condition in Gerris is Dirichlet (zero) for the normal components of the velocity and Neumann (zero) for all other variables.

Let us say that we want to impose a Poiseuille (parabolic) profile rather than a constant inflow velocity for the half-cylinder problem i.e. we want a Dirichlet boundary condition on the normal component of the velocity (`U`) with an imposed parabolic profile. This can easily be done in Gerris like this:

```
...
GfsBox { left = GfsBoundary {
                  GfsBcDirichlet U { return 1. - 4.*y*y; }
                  GfsBcDirichlet V 0
                }
       }
GfsBox {}
GfsBox {}
GfsBox { right = GfsBoundaryOutflow }
...
```

Similarly a Neumann boundary condition on variable `X` would use `GfsBcNeumann X ...`

4.2 Adding tracers

In the half cylinder example, it would be nice to be able to visualise the flow using for example a passive tracer injected at the inlet. This is very simple, just modify the `half-cylinder.gfs` parameter file like this:

```
4 3 GfsSimulation GfsBox GfsGEdge {} {
  GfsTime { end = 9 }
  GfsRefine 7
  GtsSurfaceFile half-cylinder.gts
  GfsVariableTracer {} T
  ...
  GfsOutputPPM { step = 0.02 } tracer.ppm {
    min = 0 max = 1 v = T
  }
  GfsOutputSimulation { step = 0.1 } half-cylinder-%3.1f.gfs {
    variables = U,V,P,T
  }
  ...
```

```
}
GfsBox { left = GfsBoundary {
                  GfsBcDirichlet U 1
                  GfsBcDirichlet V 0
                  GfsBcDirichlet T { return y > 0. ? 1. : 0.; }
                }
       }
...
```

which will inject tracer `T` at the inlet only in the upper half of the channel.

The adaptive refinement algorithm shoud also take your tracer into account. Try this

```
  ...
  GfsAdaptVorticity { istep = 1 } { maxlevel = 7 cmax = 1e-2 }
  GfsAdaptGradient { istep = 1 } { maxlevel = 7 cmax = 1e-2 } T
  ...
```

which will adapt using both the gradient of tracer `T` and the vorticity.

You can have any number of tracers you want, they are dynamically allocated.

4.3 Adding diffusion terms

Up to now, we have only considered inviscid, incompressible flows. Without going into the details, this type of problems require the solution of two main subproblems: solving a Poisson equation for the pressure and an advection equation for the momentum and tracers with the corresponding boundary conditions.

Gerris can also solve a third class of subproblems: diffusion equations. Diffusion equations are similar to Poisson equations (they both involve Laplacian operators) and can be solved efficiently using the same multigrid solver we use for the pressure.

In practice adding diffusion to a given tracer is as simple as adding:

```
  ...
  GfsSourceDiffusion {} T 0.01
  ...
```

to the parameter file, where 0.01 is the value of the diffusion coefficient.

4.3.1 Boundary conditions for diffusion terms

What if we want to modify the tracer example above so that now the half-cylinder itself is a (diffusive) source of tracer rather than the inlet? We need to be able to impose this boundary condition on the embedded solid surface. On embedded solids, the default boundary conditions for the diffusion equation is Neumann (zero flux) for tracers and Dirichlet (no-slip) for the velocity components. To change that use

```
  ...
  GfsVariableTracer T
  GfsSourceDiffusion {} T 0.001
  GfsSurfaceBc T Dirichlet 1
  ...
```

and change the inlet boundary condition back to

```
  ...
  GfsBox { left = GfsBoundary {
                  GfsBcDirichlet U 1
                  GfsBcDirichlet V 0
                }
       }
  ...
```

5 Writing new objects

If you want to implement your own boundary conditions, initial conditions, outputs, criterium for adaptive mesh refinement etc... you will need to learn how to write new objects within the gerris framework. As gerris is written in C, a basic knowledge of C programming will help a lot.

Gerris uses the object-oriented framework provided by GTS. This is essentially a set of structures and functions implementing the fundamental concepts of object-oriented programming (essentially data and methods inheritance and overloading) using C. Of course, as C is not an object-oriented language, it does not provide direct (syntaxical) support for this and consequently, as we will see, the expression of these concepts is rather more verbose than in, say, C++, python, smalltalk or Java.

So, why use C and not an object-oriented language? An exhaustive discussion would be too long for this tutorial. In short, the strongest case for this approach is that C is the smallest common denominator between most of the languages out there: it is possible to use the gerris library with any one of C++, Fortran 90, Java, Perl, python, ruby etc... but it would not be the case if gerris was written in any of these other languages.

5.1 A template for new object classes

GTS comes with a simple script called `gtstemplate` which will generate a C code template for the new class you want to create. For a summary of its syntax just type:

```
% gtstemplate
Usage: gtstemplate [OPTIONS] Class ParentClass
Options:
        [--no-extra-data]
        [--no-extra-method]
        [--overload=METHOD]
```

As an example we are going to try and create a template for a new initial condition class called `InitPeriodic`. Just type:

```
% gtstemplate --overload=read --overload=write --overload=event \
InitPeriodic GfsInit > init_periodic.c
```

We have just created a template for a new class called `InitPeriodic` derived from `GfsInit` and where the `read`, `write` and `event` methods are overloaded. Fire up your favourite editor and have a look at the file generated: `init_periodic.c`. The first thing you see is that the file is divided in two sections starting with

```
/* InitPeriodic: Header */
```

and

```
/* InitPeriodic: Object */
```

As their names indicate these sections correspond respectively to the declaration of the structures and functions (header part) and to the corresponding definitions (object part). If we were to use these functions and structures in a library, these two parts would be in separate files (a `.h` and a corresponding `.c` file). For what we are interested in (a gerris *module*) they are fine staying in the same file. There is a function we will not need: `init_periodic_new`, just remove the lines declaring and defining it in the header and object sections. We are left with only one function: `init_periodic_class`. This function essentially registers our new class and its associated attributes. Things like:

- the name of the class: `"InitPeriodic"`,
- the sizes of the object and class structures: `sizeof (InitPeriodic)` and `sizeof (InitPeriodicClass)`,
- function to call to initialize the class: `init_periodic_class_init`,

- function to call to initialize a new object: `init_periodic_init`,
- as well as which class is the ancestor of this new class: `gfs_init_class ()`.

If we now look at what the `init_periodic_class_init` function does, we see that it indeed overloads the `event`, `read` and `write` methods of our class with locally defined functions. The `GFS_EVENT_CLASS` and `GTS_OBJECT_CLASS` macro calls are casting operators which change the type of our new class (`InitPeriodicClass`) to the types of one of its parent class. `GtsObjectClass` is the ancestor of all object classes and we see that the `read` and `write` methods are thus defined for all objects. The `GfsInit` class is also a descendant of the `GfsEvent` class (as we have seen before it is a special type of event occurring once at the start of the simulation) which has and associated `event` method.

5.2 Initial conditions

With this preliminary overall understanding of what the template does, we are ready to define our new initialisation class. What we want to do is initialise the 2D velocity field using the following functions:

$$u(x,y) = -\cos(2m\pi x)\sin(2m\pi y) \quad (1)$$
$$v(x,y) = \sin(2m\pi x)\cos(2m\pi y) \quad (2)$$

where m is a parameter. This is an exact stationary solution of the 2D Euler equations with periodic boundary conditions.

We need to store the value of m somewhere. A clean way to do that is to add m as a data associated with our `InitPeriodic` object. To do this just modify the definition of the structure in the header part of the file, like this:

```
struct _InitPeriodic {
  /*< private >*/
  GfsInit parent;

  /*< public >*/
  gdouble m;
};
```

Note that the `GfsInit parent;` data *must be the first* data in the structure. The whole object-inheritance mechanism in C relies on this data alignment.

We now need to specify the value of m. The first value we want to assign is the default value of m for an object newly created. We can do that using the initialisation function `init_periodic_init` which is called everytime a new `InitPeriodic` object is created (it is the *constructor* function of the `InitPeriodicClass`). Just add:

```
static void init_periodic_init (InitPeriodic * object)
{
  object->m = 1.;
}
```

Our default value for m is now one. What we really want is being able to specify the value of m in the parameter file used by gerris. We can do that by using the `read` and `write` methods. We will start with the `write` method: `init_periodic_write`. This function first calls the `write` method of the parent class of our object:

```
  /* call write method of parent */
  if (GTS_OBJECT_CLASS (init_periodic_class ())->parent_class->write)
    (* GTS_OBJECT_CLASS (init_periodic_class ())->parent_class->write)
      (o, fp);
```

The parent class is given by the `parent_class` field of any `GtsObjectClass`. The `write` method might not be defined for the parent class, so it is always a good (safe) idea to first test that it is indeed defined.

Now that we have written the data associated with the parent class, we can write our own data like this:

```
static void init_periodic_write (GtsObject * o, FILE * fp)
{
  /* call write method of parent */
  if (GTS_OBJECT_CLASS (init_periodic_class ())->parent_class->write)
    (* GTS_OBJECT_CLASS (init_periodic_class ())->parent_class->write)
      (o, fp);

  fprintf (fp, " %g", INIT_PERIODIC (o)->m);
}
```

Note that we used a space to separate this new data from the data of the parent class and that we didn't add anything after our own data (no space or newline). This is so that we can eventually extend (through inheritance) this class by adding more data in exactly the same way.

We now need to define a "symmetrical" `read` method which will read the value of m from the parameter file. In the `init_periodic_read` method, we first read the parameters associated with the parent class. The line:

```
  if (fp->type == GTS_ERROR)
    return;
```

just checks if an error occurred while reading parameters for the parent class, in which case the method returns prematurely. To read our parameter we are going to use functions associated with the `GtsFile` data type (have a look at the GTS reference manual[21] for details):

```
static void init_periodic_read (GtsObject ** o, GtsFile * fp)
{
  /* call read method of parent */
  if (GTS_OBJECT_CLASS (init_periodic_class ())->parent_class->read)
    (* GTS_OBJECT_CLASS (init_periodic_class ())->parent_class->read)
      (o, fp);
  if (fp->type == GTS_ERROR)
    return;

  if (fp->type != GTS_INT && fp->type != GTS_FLOAT) {
    gts_file_error (fp, "expecting a number (m)");
    return;
  }
  INIT_PERIODIC (*o)->m = atof (fp->token->str);

  /* do not forget to prepare for next read */
  gts_file_next_token (fp);
}
```

We first check that the current token is an integer (`GTS_INT`) or a floating point number (`GTS_FLOAT`). If it is not we set an error message describing the problem and return prematurely. If it is, we convert the string describing the current token (`fp->token->str`) into a floating point number (using the standard C `atof` function) and assign it to m. Note that we need to use the type conversion macro `INIT_PERIODIC` because the argument passed to `init_periodic_read` is a generic `GtsObject`. As we are dealing with a method of our `InitPeriodicClass` we know that this `GtsObject` is also a `InitPeriodic` object and the type conversion is then legitimate.

[21] http://gts.sf.net/reference/book1.html

We can now create, read and write our new object. However we are not doing anything with it yet. Like for any other `GfsEvent`, the action performed is controlled by the `event` method. If we look at the `init_periodic_event` function, we see that it returns a `gboolean`. If `TRUE` this return value means that the event took place. We then call the event method of the parent class and check its return value. If `TRUE` we do something and return `TRUE`. We know that the parent class is `GfsInit`, an event which takes place once at the start of the simulation. What we want to do is initialise the velocity field using our formula. We can do that like this:

```
static void init_velocity (FttCell * cell,
                           InitPeriodic * init)
{
  FttVector pos;

  ftt_cell_pos (cell, &pos);
  GFS_STATE (cell)->u =
    - cos (2.*init->m*M_PI*pos.x)*sin (2.*init->m*M_PI*pos.y);
  GFS_STATE (cell)->v =
    sin (2.*init->m*M_PI*pos.x)*cos (2.*init->m*M_PI*pos.y);
}

static gboolean init_periodic_event (GfsEvent * event, GfsSimulation * sim)
{
  if ((* GFS_EVENT_CLASS (GTS_OBJECT_CLASS (init_periodic_class ()))\
        ->parent_class)->event) (event, sim)) {
    gfs_domain_cell_traverse (GFS_DOMAIN (sim),
                              FTT_PRE_ORDER, FTT_TRAVERSE_LEAFS, -1,
                              (FttCellTraverseFunc) init_velocity,
                              event);
    return TRUE;
  }
  return FALSE;
}
```

The `gfs_domain_cell_traverse` function traverses the cell tree and calls the `init_velocity` function for each leaf cell (`FTT_TRAVERSE_LEAFS`). A `GfsSimulation` is an object derived from `GfsDomain` which justifies the `GFS_DOMAIN (sim)` type casting. Have a look in the reference manual[22] if you want to know more about these structures and functions.

We also pass an extra parameter to `init_velocity`: `event`. If you now look at `init_velocity` you see that this extra argument is our object: `InitPeriodic * init`. What we have done here is an implicit cast of `GfsEvent * event` to `InitPeriodic * init`. We know it is valid because `init_periodic_event` is a method of our object class.

What happens in `init_velocity` is straightforward. We first get the position of the center of the cell using `ftt_cell_pos` and then assign the values of the two components of the velocities using our formula and the m parameter defined in `init`.

5.2.1 Creating a module

We have now almost all that is needed for our new object. How do we use this new piece of code with gerris? What we want to do is to create a dynamically loadable `module`. First of all we need to check that we can compile the code. There are a few missing headers we need to add (at the top of the file):

```
#include <math.h>
#include <stdlib.h>
#include <gfs.h>
```

[22] http://gfs.sf.net/reference/book1.html

The `gfs.h` header contains all the function declarations for the gerris library. We can now try to compile using for example:

```
% cc `gfs-config --2D --cflags` -Wall -g -O2 -c init_periodic.c
```

where `gfs-config --2D --cflags` (quoted using inverted quotes) defines the compilation flags needed to use the 2D version of the gerris library.

To make a proper module we also need to add the following at the end of the file:

```
/* Initialize module */

const gchar * g_module_check_init (void);

const gchar * g_module_check_init (void)
{
  init_periodic_class ();
  return NULL;
}
```

This just tells gerris how to initialise the module after it has been loaded. The `g_module_check_init` function will be called. In our case, it does only one thing which is to *instantiate* our new class: this is necessary so that gerris registers how to handle it.

We are now all set to create our new module. Sadly, dynamically loadable module creation is not a standardised process and the command-line arguments vary from compiler to compiler and from system to system. In the following, I will assume that you use `gcc` on a linux box. If you are using another system supporting dynamically loadable modules, you will need to read your local manual. On a linux box:

```
% cc `gfs-config --2D --cflags` -Wall -g -O2 -c -fPIC init_periodic.c
% cc -shared init_periodic.o -lc -o libinit_periodic2D.so
```

should work. Note that we add the `2D` extension to indicate that this module uses the 2D gerris library. We could have built both a 2D and a 3D version of the same module. At runtime the gerris executable uses this extension to check which version to load.

Our module is now ready to use. We just need to install it in a directory where it will be found by gerris. If you installed gerris in `/home/joe/local` just type:

```
% cp libinit_periodic2D.so /home/joe/local/lib/gerris
```

We can now use it directly in a parameter file, for example:

```
1 2 GfsSimulation GfsBox GfsGEdge {} {
  GfsTime { end = 50 }
  GfsRefine 6
  GModule init_periodic
  InitPeriodic {} 2
  GfsOutputTime            { istep = 10 } stdout
  GfsOutputProjectionStats { istep = 10 } stdout
  GfsOutputPPM             { step = 0.1 } vorticity-%4.2f.ppm { v = Vorticity }
}
GfsBox {}
1 1 right
1 1 top
```

The first part of the object definition: `InitPeriodic {}` is the generic `GfsEvent` definition, the second part: `2` is the parameter m we added. What if we do not specify the right parameter? Just try to replace `InitPeriodic {} 2` with `InitPeriodic {} a`. You should get a message like

```
gerris: file 'periodic.gfs' is not a valid simulation file
periodic.gfs:5:18: expecting a number (m)
```

which tells you where the error occurred (in file `periodic.gfs`, line 5, character 18) together with the error message which we specified in the `read` method of our class.

5.3 Outputs

5.4 Boundary conditions

6 Running Gerris in parallel

7 Learning more

While this tutorial should give you a good overview of Gerris, it is by no means a complete description. To learn more you should first consult the Gerris Frequently Asked Questions[23] and the Gerris reference manual[24] which describes each object and the corresponding file parameters in more detail.

You should also have a look at the Gerris Examples[25] page for examples of how to use Gerris for a range of problems. The parameter files are cross-linked with the reference manual.

Another source of more advanced examples is the Gerris test suite[26].

If things are still unclear you can ask for help on the `gfs-users` mailing list[27]. Please note that you first need to subscribe to the list to be able to post messages.

8 Do you want to help?

The idea behind Gerris and other free software projects is that transparency, free exchange of information and cooperation benefit individuals but also society as a whole. If you are a scientist, you know that these same principles are also keys to the efficiency of Science.

Helping with Gerris development can be done in various ways and aside from giving you this altruistic, warm fuzzy feeling of helping others will also benefit you directly. A few concrete simple ways of helping are (in approximate order of difficulty):

- Use the code, comment on the problems you find, what you like, don't like about it.
- Share your results with other Gerris users, write a web page about the problem you solved using Gerris etc...
- If you publish papers using Gerris, send me the reference. It is very useful to be able to show evidence of wider usage when seeking continued funding for the project.
- Also, if Gerris capabilities are central to your article feel free to ask me to be a co-author on your paper...
- Have a look at the Gerris internals (write your own modules) and share them with us.
- Think of ways to extend Gerris for your own problems, implement them and share them with us (you can count on my and other developers' help).

[23] http://gfs.sf.net/faq/faq/faq.html
[24] http://gfs.sf.net/reference/book1.html
[25] http://gfs.sf.net/examples/examples
[26] http://gfs.sf.net/tests/tests/index.html
[27] http://gfs.sf.net/mailinglists.html