Free Computational Fluid Dynamics

STÈPHANE POPINET

What is Gerris? Named after the *Gerris remigis* or common water strider, Gerris is yet another tool for computational fluid dynamics. Three years ago, I started working on a new numerical technique for the solution of flows with fluid/fluid interfaces (e.g. droplets, bubbles, waves etc.) in three dimensions. I had several choices: I could carry on using the 2D and 3D research codes I had used and developed for several years, I could try to find a (preferably cheap) well-designed, extensible commercial code, or I could design a new code from scratch trying to avoid the pitfalls I had fallen (or been drawn) into with my previous codes.

After some research, it became clear that the second option (commercial code) was going to be a no-starter: commercial codes I had worked with or knew of were prohibitively expensive, even more so when access to some of the internal source code was required. Most of them also fell short of the well-designed, extensible requirement. The "easiest" option would then have been to carry on using the research code I was had been using, adding yet another layer of complexity on an already shaky structure.

As happens often to research codes, this program had been initially designed as a simple, efficient implementation of a restricted class of new algorithms but had grown into a multi-headed monster: compilation switches were used to select various versions, lack of structure created function calls with an ever-increasing number of parameters, etc... Despite this complexity, the code was also quite limited in capabilities: it had good support for the interfacial flows it had been designed for, but could not deal with complex solid boundaries and was using a simple regular Cartesian grid. Trying to extend it yet again was clearly a recipe for future disaster.

Ideally, I saw the new code as combining the best qualities of good commercial offerings: robustness, wide range of applications; and good research codes: cutting-edge methods, efficiency, openness; and of course, without the drawbacks (and yes, this is pretty ambitious). More concretely, I set the following design goals:

- Generic: 2D or 3D, simple and complex boundaries, single or multiphase flows
- Extensible: re-usable library design (with language-agnostic bindings), dynamically-linked extension modules
- Portable and parallel: programmed in C, MPI
- Powerful and easy to use: self-contained, flexible parameter files, no-need for code re-compilation
- Cutting-edge and efficient: it is a research code

After three years of development, I believe Gerris is on track to fulfill these targets. It is significantly different from existing commercial and research codes. First of all, it is really "free," "as in beer" but more importantly "as in speech." With a few remarkable exceptions, like the Mouse package, most freely available research codes are distributed under restrictive licenses, often due to US-government policies but sometimes also as a manifestation of the desire of researchers to keep a tight control on their work. While competition is undoubtedly an important aspect of science, science would do very little without collaboration and the free exchange of ideas (see *www.lmm.jussieu.fr/~zaleski/OpenCFD.html*). In this respect, I consider freely releasing source codes as just another essential form of scientific publication. From a practical point of view, the incentive to use a "free as in beer" research code for one's scientific work is not much larger than using a commercial code: the same lack of control over the future of the code and one's contributions to it applies. Using a "free as in speech" license guarantees the perennity of the code and empowers users with a degree of control over the development process. From a business perspective, it also makes a lot of sense for an organization to benefit from the expertise of a community of researchers to develop a tool which can then be used for core consulting activities.

Gerris is also different from a technical point of view. It is really a research code in the sense that it is a unique

combination of recent new approaches in CFD. While most modern solvers are based on fully-unstructured meshes, Gerris uses semi-structured quadtree/octree meshes. This type of discretization is a good compromise between the flexibility of unstructured meshes and the computational efficiency of structured meshes. The main problem with this approach is that the discretized elements cannot generally be made to follow the boundaries of complex domains. To overcome this difficulty, Gerris uses an embedded solid boundary representation where cells cut by solid boundaries are treated differently in order to apply boundary conditions accurately. An important practical consequence of this approach is that the often difficult and time-consuming first step of domain meshing (for unstructured meshes) can be entirely automated. Better still, the quad/octree mesh creation process is efficient enough to be applicable at every time step with little penalty. The solver can then adapt the mesh dynamically to the solution being computed. This result can lead to huge improvements in computation time and/or solution quality.

Current Capabilities

Time-dependent incompressible flows are the primary focus. *Figure One* illustrates an example of 3D Euler incompressible turbulent air flow around the research vessel Tangaroa of NIWA. The time-adaptive mesh of *Figure Two* is used to discretize the solution dynamically. We also validated the numerical results using in-situ wind measurements. The meshed CAD model of the vessel was imported directly into Gerris and used for automatic mesh generation. The code can also solve variable-density flows, advection-diffusion equations (e.g. passive tracers). Two-phase flows can be treated using a Volume Of Fluid (VOF) advection scheme in 2D and 3D. Stokes and Navier-Stokes flows are for the moment restricted to 2D.

An Example

After this general overview, let's look at how Gerris works in practice.

I will first assume that you have installed all the required components (see the Sidebar).

As a first example we will setup a Navier-Stokes simulation of a 2D flow around a circular cylinder. The flow will be from left to right, with the cylinder placed in a straight channel eight times longer than it is wide. A passive tracer will be injected at the inlet to visualize the flow. We will also use dynamic adaptive mesh refinement.

Gerris is a command line-oriented program. The code is controlled through a parameter file. One of the design goals was that every aspect of the simulation should be controllable through the parameter file without any need for recompilation or tinkering within the code.

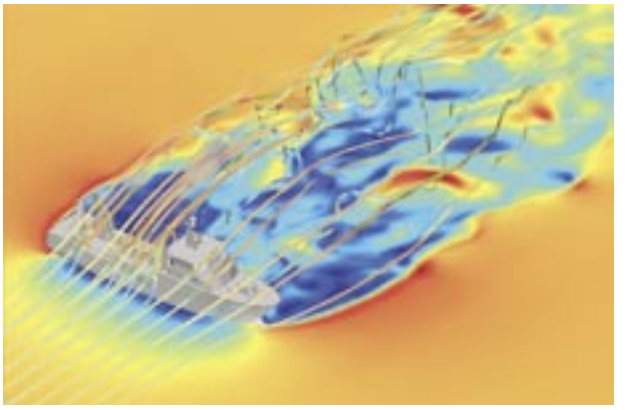

FIGURE ONE Snapshot of the simulation of a 3D air flow around a CAD model of the research vessel Tangaroa. The colors represent the local amplitude of the wind speed.

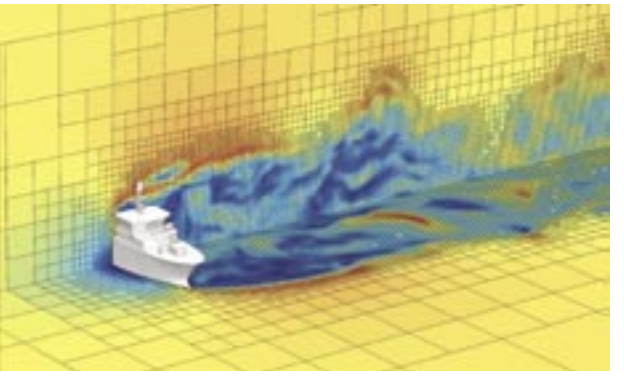

FIGURE TWO Vertical and horizontal cross sections showing the adaptive mesh used to resolve the flow

Let's first define the simulation domain. Fire up your favorite text editor and copy the following:

```
8 7 GfsSimulation GfsBox GfsGEdge {} {
}
GfsBox {}
GfsBox {}
GfsBox {}
GfsBox {}
GfsBox {}
GfsBox {}
GfsBox {}
GfsBox {}
1 2 right
2 3 right
3 4 right
4 5 right
5 6 right
6 7 right
7 8 right
```

This defines the computation domain as a graph (of type `GfsSimulation`) of eight nodes (of type `GfsBox`) linked by seven edges (of type `GfsGEdge`). Each of the nodes is then defined (the eight lines starting with `GfsBox`) and each of the edges (the seven lines ending with `right`). The edges are simply defined by the indexes of the nodes they are connecting together with the direction in which the connection is done (right, left, top, bottom). All the blocks delimited by braces are placeholders for optional parameters. A `GfsBox` is simply a square domain (cube in 3D) of size unity. This parameter file thus defines a rectangular domain of size 8x1.

What about boundary conditions? If left unspecified, the default is a "mirror" or symmetry condition for all fields. That will do fine for the sides of the channel (a friction-free solid wall) but we want to impose a constant inflow velocity on the left-side of the channel and an outflow condition on the right side. We can do that easily by adding the following arguments to the first and last nodes:

```
8 7 GfsSimulation GfsBox GfsGEdge {} {
}
GfsBox {
  left = GfsBoundary {
    GfsBcDirichlet U 1
  }
}
...
GfsBox { right = GfsBoundaryOutflow }
```

This defines the right side of the last node as a `GfsBoundaryOutflow` and the left side of the first node as a generic `GfsBoundary`. A `GfsBoundary` takes extra optional arguments defining the boundary conditions variable by variable. The `GfsBcDirichlet U 1` line sets this left-side boundary as a Dirichlet (fixed value) boundary condition of `1` for the `U` variable (the horizontal component of the velocity). We also want to inject some tracer at the inlet. This can be done by adding a boundary condition for a `C` variable, like this:

```
...
  left = GfsBoundary {
    GfsBcDirichlet U 1
    GfsBcDirichlet C { return y < 0. ? 1. : 0.; }
  }
...
```

This is similar to the boundary condition for `U` but here we have used a powerful feature of the parameter file system: the possibility to specify most numeric arguments as functions of space (x,y,z) and time t. The functions are written as C functions (you will thus need some basic notion of the C language to use them). By convention the first `GfsBox` of the graph is centered on the origin (x = y = 0). The boundary condition we wrote just sets `C` to `1` in the lower half of the boundary (y < 0.) and to zero otherwise.

For the moment we have not told the code how to discretize our domain (how many grid points we want and so on), for how long the simulation should run and what the initial conditions are. This can be done by adding:

```
8 7 GfsSimulation GfsBox GfsGEdge {} {
  GfsTime { end = 15 }
  GfsRefine 6
  GfsInit {} { U = 1 }
}
...
```

The argument to `GfsRefine` is the number of quadtree levels the code should use to discretize each `GfsBox`. In our case 6, which will give 2^6=64 grid points in the width of the channel. This argument could also be a function of space if we wanted a non-uniform mesh. The line starting with `GfsInit` specifies the boundary conditions for each variable. In our case we just set `U` to `1` everywhere in the domain, all the other variables are zero by default.

This file is enough to run a (pretty boring) simulation: uniform flow in a channel and more problematically no outputs! Outputs are controlled by a number of objects all derived from a `GfsOutput` parent class. Several objects are provided with Gerris to output various things one may be interested in. For advanced users, it is also easy (with some knowledge of C programming) to define one's custom new output object with very few limitations on what can be done. Let's add a few to our parameter file:

```
8 7 GfsSimulation GfsBox GfsGEdge {} {
  GfsTime { end = 15 }
  GfsRefine 6
  GfsInit {} { U = 1 }
  GfsOutputTime { istep = 10 } stdout
  GfsOutputBalance { istep = 10 } stdout
  GfsOutputProjectionStats { istep = 10 }
    stdout
  GfsOutputPPM { istep = 2 } vort.ppm {
    min = -10 max = 10 v = Vorticity
  }
  GfsOutputPPM { istep = 2 } c.ppm {
    min = 0 max = 1 v = C
  }
```

```
  GfsOutputTiming { start = end } stdout
}
...
```

All output objects share a similar structure. The first optional argument defines the time window and recurrence of the output "event". Units are either physical time or number of time steps taken. For example `istep = 2` means "output every second time step" while `step = 2` would mean "output every two physical time units." Similarly `start = end` means "execute this only once at the end of the simulation." The second argument tells the code where to write the resulting output, it can be a file name or a pipe (more on this later). The file names `stdout` and `stderr` have special meaning as the standard output and standard error respectively. `GfsOutputTime` writes the current iteration and physical time, `GfsOutputBalance` gives statistics on the number of grid points, `GfsOutputProjectionStats` statistics on the convergence rate of the solver. `GfsOutputPPM` is more interesting. It produces a PPM (Portable PixMap) bitmap image of the field given by the `v =` parameter.

There are still three things missing from our parameter file: the cylinder, adaptive mesh refinement and fluid viscosity. To define solid boundaries, Gerris needs a "triangulated surface" as input: that is, a surface mesh composed of triangular elements. This mesh needs to be a "proper" surface: i.e. it must be closed (no gaps, however small), it must not be folded on itself, etc... This is very much the same type of requirement as for most unstructured mesh generators. Unfortunately, many Computer Aided Design (CAD) packages do not always produce well-behaved surfaces (they may look good on screen but are full of tiny cracks and folds). There consequently is a whole industry of software dealing purely with "mesh repair". For our example, I will just provide you with a proper triangulated mesh of a cylinder (see Resources). Gerris expects files to be in the GTS (GNU Triangulated Surface) format, a simple text format. The distribution also comes with a few conversion utilities (such as `dxf2gts` which converts AutoCad triangulated surfaces to the GTS format). To add this solid boundary to the simulation just copy the GTS file to the working directory and add the following to the parameter file:

```
...
  GtsSurfaceFile cylinder.gts
...
```

Adaptive refinement and viscosity are controlled by adding

```
...
  GfsAdaptVorticity { istep = 1 } {
    maxlevel = 6 cmax = 1e-2 }
  GfsAdaptGradient { istep = 1 } {
    maxlevel = 6 cmax = 1e-2 } C
  GfsSourceViscosity {} U 0.00078125
...
```

We use two criteria to control mesh adaptation: the vorticity (i.e. "rate of rotation") of the flow field and the gradient of the passive tracer `C`. In both cases adaptation is performed at every timestep `{ istep = 1 }`, the maximum number of refinement levels allowed is 6 (if unspecified the minimum level is 1). The refinement stops whenever the criterion becomes smaller than `cmax`.

The viscosity nu we set corresponds to a Reynolds number Re=UD/nu of 160. Where U is the inflow velocity (1 in our case), D is the diameter of the cylinder (1/8 of the channel width).

Everything is now ready, just save the file as `cylinder.sim` and on the command line type:

```
$ gerris2D cylinder.sim
```

If you have not made any typos, the code should run, writing information on the screen every ten time steps. If in another terminal you now type

```
$ animate vort.ppm
```

or

```
$ animate c.ppm
```

you will be able to follow graphically the progress of the simulation (note that `animate` does not automatically re-read the PPM file as the simulation progresses, you need to restart it).

What you will find is that the PPM files grow quite quickly. The PPM file format is portable and very simple but no compression at all is used which makes it quite inefficient as a storage format (even more so for movies). Generating an MPEG movie would be nice. One solution would be to wait for the end of the simulation and then convert the PPM files to MPEG files using some conversion tool. A better solution (which will not generate large temporary files) is to do the conversion "on the fly." Using the powerful "pipe" feature of the parameter file and the right conversion tool (MJPEG Tools) it is very easy to do. Just change the relevant lines in the parameter file to:

```
...
  GfsOutputPPM { istep = 2 } { ppmtoy4m -F
    24:1 -v 0 | mpeg2enc -v 0 -o vort.mpg } {
    min = -10 max = 10 v = Vorticity
  }
  GfsOutputPPM { istep = 2 } { ppmtoy4m -F
     24:1 -v 0 | mpeg2enc -v 0 -o c.mpg } {
    min = 0 max = 1 v = C
  }
...
```

`GfsOutputPPM` will do the same thing as before but instead of directly writing the result to a file, it will pipe the PPM image into the `ppmtoy4m` and `mpeg2enc` commands (themselves connected by a pipe) and directly generate a MPEG file. This is just an example of how a running Gerris simulation can be connected to various post-processing tools, visualization etc. ...

With this improvement done, you probably want to stop the running simulation (`Ctrl-c` will do) and rerun it with the improved parameter file. The MPEG files generated can then be visualized using your favorite player.

If you are patient (15 to 30 minutes waiting time) you will get to the stage of the fully developed Von Karman vortex street pictured in *Figure Three*.

More on post-processing

For the moment we have only used very simple built-in "post"-processing of the simulation results. There are several other post-processing options built into the Gerris code base but for more sophisticated visualizations (particularly in 3D), it is better to use other packages dedicated to visualization. Unfortunately, all the packages I have tested are systematically geared toward either structured (Cartesian-like) or fully-unstructured meshes (triangular or quadrilateral elements in 2D, tetrahedra in 3D) and thus do not take full advantage of the multiscale representation automatically given by the semi-structured approach used by Gerris (see sidebar for details).

To use an external visualization tool, the first step is to save the results of the simulation for the time steps we are interested in. This can be done by adding:

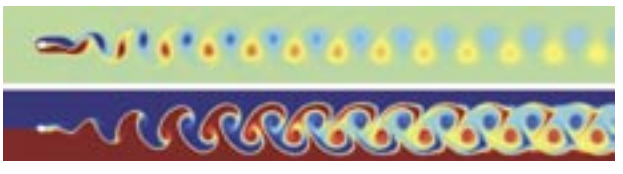

FIGURE THREE Von Karman vortex street behind a cylinder. Vorticity field (top) and passive tracer (bottom).

```
...
  GfsOutputSimulation { start = 1 step = 1
    } cylinder-%02.0f.sim {}
...
```

If you then restart the simulation, it will now save files called `cylinder-01.sim`, `cylinder-02.sim`, ..., `cylinder-15.sim`. These files contain all that is necessary to restart the simulation. If you open one in your text editor you will see that it is very similar to your cylinder.sim parameter file apart from a large section containing the detail of the discretization and values of the variables. You can of course change some of the parameters, add outputs etc... and restart the simulation from where it stopped with the updated file.

Gerris comes with several filters that can convert simulation files to formats readable by external visualization programs. For development I mostly use a slightly crufty filter called `gfs2oogl` in combination with Geomview. Geomview is a nice generic 3D object viewer with a number of interesting features but is not geared specifically toward scientific visualization. Most of the work is thus done by the filter itself. As an example just try the following:

```
$ gfs2oogl2D < cylinder-15.sim >
    cylinder-15.oogl
$ geomview cylinder-15.oogl
```

With some panning and zooming you should get an reasonable image. Gfs2oogl can do much more than this but is not very user-friendly: try `$ gfs2oogl2D -h` for a brief overview of the various options.

For a more graphical experience, you can try the excellent MayaVi, which is geared specially toward flow visualization. It can do classical things like cross-sections, isosurfaces, vector plots etc... The `gfs2vtk` filter can be used to generate VTK (Visualization Tool Kit) files readable by MayaVi. Try the following:

```
$ gfs2vtk2D cylinder-15 < cylinder-15.sim
$ mayavi -d cylinder-15_field.vtk
```

Click on the "Configure Data" button, select scalar "C" on the pop-up window, click on the "Visualize -> Modules -> SurfaceMap" menu entry and you should get your first Mayavi visualization.

For a very powerful but relatively heavy graphical visualization tool, you can try IBM's open-source OpenDX. This is more a graphical programming environment (modules linked graphically by pipes) than a

stand-alone visualization tool. It is also very powerful. Gerris comes with a special OpenDX module that can be used to import simulation files directly into OpenDX. To use it, start OpenDX like this:

```
$ dx -mdf /usr/local/lib/gerris/dx2D.mdf
```

If you are not familiar with OpenDX you should then read the extensive documentation.

Parallel Computation

Gerris was designed from scratch with parallel computation in mind. I chose a simple domain decomposition approach, which is one of the main reason of the slightly weird graph structure of the domain definition in parameter files.

Let's come back to our 2D example. Suppose you want to use 8 processors of your parallel machine. For the domain we have, a fairly obvious decomposition strategy is just to assign each of the eight GfsBox to each processor. This can be done easily by modifying the parameter file like this:

```
...
GfsBox { pid = 0 }
GfsBox { pid = 1 }
GfsBox { pid = 2 }
GfsBox { pid = 3 }
GfsBox { pid = 4 }
GfsBox { pid = 5 }
GfsBox { pid = 6 }
GfsBox { pid = 7 }
...
```

The simulation can then be ran on eight processors like this:

```
$ mpirun -np 8 gerris2D cylinder.sim
```

The simulation file can still be run on a single processor as usual (the `pid` info is just ignored).

What is currently lacking is a parallel output capability. The files written by each processor are entirely independent (i.e. they represent only the part of the domain simulated by this processor). It is then up to you to put the puzzle back together.

What we did is an example of manually partitioning the domain. In more complex cases, Gerris can also automatically partition the domain for a given number of processors, variable resolution, complex boundaries etc. It does this using sophisticated I<graph partitioning> algorithms which aim to minimize the cost of communications between sub-domains. This is done in two steps. Let's say we want to now run our simulation on 32 processors. Just type:

```
$ gerris2D -p 5 cylinder.sim > cylinder-32.sim
```

This will create a new simulation file which can the be run on 32 processors. The `-p` option is the power of two processors, in this case two to the fifth power.

Apart from parallel outputs, the main issue which is not currently addressed by Gerris is dynamic load-balancing which is necessary to keep good parallel efficiency when adaptive mesh refinement is used.

Resources

Gerris: gfs.sf.net
GTS: gts.sf.net
Glib: www.gtk.org
Mouse: www.vug.uni-duisburg.de/MOUSE
Triangulated mesh of a cylinder: gfs.sourceforge.net/clusterworld
OpenDX: www.opendx.org
Mayavi: mayavi.sf.net
Geomview: www.geomview.org
ImageMagick: www.imagemagick.org
MJPEG Tools: mjpeg.sf.net

The Future of Gerris

As you have probably found out by now, while Gerris seems to be on the right track there is still a lot to be done to take full advantage of its flexibility. Together with collaborators at NIWA and elsewhere, I am currently working on several aspects of the code. The first short-term goal is to extend the Navier-Stokes and diffusion equations solution to three dimensions. This is made slightly more difficult by the treatment of embedded Dirichlet boundary conditions in three-dimensions.

Another active area is the development of non-newtonian viscous stresses (i.e. fluids for which the stress/strain relationship is not linear): applications include the numerical simulation of visco-plastic fluids (e.g. snow and mud avalanches, granular materials, plastics) and Large-Eddy-Simulations (LES) turbulence modeling.

A very recent endeavor has been the extension of Gerris to geophysical fluid dynamics i.e. oceanic and atmospheric flows, where the adaptive mesh approach has proved extremely useful.

Potential extensions which would make full use of the existing framework are numerous. Just a few ideas including; steady-state flow solvers, compressible

flows, and Python bindings.

In addition, the hierarchical quadtree/octree meshes used by Gerris allows for a true multi-scale representation of the solution. Visualization would greatly benefit from this, particularly for large 3D simulations. Unfortunately, I am not aware of any existing visualization tool designed to take advantage of this. If you like writing visualization programs and want to contribute a cutting-edge visualization tool, you will gain the eternal gratitude of Gerris users!

Whether all or none of these extensions happen depends on the community of Gerris users. I hope that you have found Gerris appealing and that you will consider contributing the missing features you wish it had. And remember, the beauty of (really) free software is that whatever happens, your contribution will always be there for you (and others) to share.

Acknowledgments

Gerris was made possible by the support of the National Institute of Water and Atmospheric research (NIWA) and the Marsden Fund of the Royal Society of New Zealand.

StÈphane Popinet is a research scientist at the National Institute of Water and Atmospheric research, Wellington, New Zealand. He can be reached at s.popinet@niwa.cri.nz.

Installing Gerris

Gerris is written in C and uses two other C libraries which you need to install on your system first; the Glib library and the GTS library. If you want to run the parallel version of Gerris, you will also need some implementation of the MPI (Message Passing Interface) library.

Gerris is quite portable and has been tested on Linux clusters, CRAY T3E parallel computer, HP and Silicon Graphics workstations. 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
```

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.

You now 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:

```
$ ./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 probably want to add `/home/joe/local/bin` to your `PATH` and `/home/joe/local/lib` to your `LD_LIBRARY_PATH`. The next step is installing Gerris itself, do the same, go to the Gerris web site 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. We are now ready to start. Just to check that everything is working try:

```
$ gerris2D -V
```

For visualization you probably also want to use the ImageMagick tools (which are installed by default on most Linux distributions) as well as Geomview, Mayavi or OpenDX. We will also make use of the MJPEG Tools for the generation of MPEG movies.

Note that Gerris will compile and install the OpenDX modules only if OpenDX was installed on your system before compiling Gerris.