arb

code version 0.25

Dalton Harvie

February 23, 2011

1 What's new?

22/2/11 Installation instructions have been improved.

3/2/11 Work has been done to make error messages more helpful.

12/12/10 The GNU GPL'd sparse linear solver UMFPACK written by Tim Davis is now interfaced to arb. An easy download and setup script is included.

25/11/10 The run procedure has been simplified for version 0.24. Basically you don't have to run `setup_equations` separately anymore - all meta-programming, compilation and running is handled by the single command './arb', run from the working directory.

2 What is arb?

arb solves arbitrary partial differential equations on unstructured meshes using the finite volume method. The code is written in fortran95, with some meta-programming done in perl with help from maxima.

The primary strengths of arb are:

- All equations and variables are defined using 'maths-type' expressions written by the user, and hence can be easily tailored to each application;
- All equations are solved simultaneously using a Newton-Raphson method, so implicitly discretised equations can be solved efficiently; and
- The unstructured mesh over which the equations are solved can be componsed of all sorts of convex polygons/polyhedrons.

arb requires a UNIX type environment to run, and has been tested on both the Apple OsX and ubuntu linux platforms. Certain third party programs are used by arb:

- A fortran compiler; ifort and gfortran are supported;
- The computer algebra system maxima;
- A sparse matrix linear solver: UMFPACK, pardiso and (currently a single) Harwell Subroutine Library routines are supported; and
- The mesh generator gmsh.

Further details about how to get and install this software are given in section 3.1.

3 Installation

3.1 Installing prerequisit software

3.1.1 Minimum setup steps required for ubuntu 10.04:

1. Install the arb files as per section 3.2
2. Install required packages:

```
sudo apt-get install maxima maxima-share gfortran
  liblapack-dev libblas-dev gmsh
```

3. Install the suitesparse solver as per section 3.1.6
4. Run a test simulation as per section 4

On ubuntu 8.04 the versions of `gfortran` and `gmsh` in the standard repositories are too old to be installed via this method.

3.1.2 Minimum setup steps required for OsX:

Installation on OsX is straightforward but takes more time.

1. Install the arb files as per section 3.2
2. Install Apple's Xcode developer package.
3. Install a gfortran binary package as per section 3.1.4
4. Install a maxima binary package as per section 3.1.3
5. Install the suitesparse solver as per section 3.1.6
6. Install the gmsh package as per section 3.1.10
7. Run a test simulation as per section 4

3.1.3 Maxima

Equation generation is performed using the Maxima Computer Algebra system. It is released under the GNU General Public License (GPL) and is available for free. To check whether you have it installed already try typing `maxima`. If the program is installed you will enter a symbolic maths environment. Then check that the command `load(f90);` finds these libraries, and then quit using `quit();`. If you need to install it:

Installing `maxima` on ubuntu:

On ubuntu linux `maxima` and the f90 package can be installed using

```
sudo apt-get install maxima maxima-share
```

Installing `maxima` on OsX using a binary package:

A precompiled version of maxima for the mac is available from sourceforge. This is a fastest way of getting things going. Download the package and copy the application 'Maxima.app' to your
`Applications` directory as directed. To make it available from the command line place the script `misc/maxima_OsX/maxima` somewhere in your path (for example in a `bin` directory).

Installing `maxima` on OsX using fink/macports:

On OsX `maxima` can be installed using either macports or fink package managers. For fink enable the unstable branch and use

```
sudo fink install maxima
```

For macports replace `fink` with `port` in the above. Compilation will take some time.

3.1.4 A fortran compiler and the blas/lapack libraries

Two different compilers have been tested with arb: `ifort` (Intel) and `gfortran`. The Intel Fortran compiler `ifort` is probably the faster of the two. It also includes the Intel Maths Kernel library which itself includes the excellent Pardiso linear solver routines (see section below): however this compiler is not free except on

linux and even then, only under specific non-commercial circumstances. The GNU compiler `gfortran` is an easier option to get going, and is freely available on both the OsX and linux platforms. It does not include the Pardiso routines but with the new interface to the UMFPACK routines this isn't a tremendous disadvantage.

Compiler choice is made automatically when the `arb` script is run (defaulting to `ifort` if it exists, otherwise using `gfortran`). These defaults can be overwritten with the `arb` options `--compiler-gnu` or `--compiler-intel`.

Installing `ifort`:

The non-commercial download site for ifort on linux is here. For OsX the compiler must be bought. Make sure you install both the compiler and MKL (Math Kernel Libraries). `ifort` versions of 11.1.069 and newer (not version 2011 yet) have been tested on both linux and OsX.

Installing `gfortran` on ubuntu:

On ubuntu linux `gfortran` and the lapack/blas libraries can be installed using

```
sudo apt-get install gfortran liblapack-dev libblas-dev
```

`gfortran` version 4.2.4 on ubuntu 8.04 doesn't seem to work on some computers (internal compiler error) whereas version 4.4.3 on ubuntu 10.04 does. On ubuntu 8.04 you could try downloading a newer binary version of `gfortran` but I haven't tried this.

Installing `gfortran` on OsX using a binary package:

The easiest way to install an up-to-date version of `gfortran` is via a precompiled package. One that worked for me is found here - currently (23/2/11) this is version 4.6. Alternatively this hpc site and this gcc site gives information about other precompiled versions. Check that once installed the `gfortran` executable is in your path - that is, typing `gfortran` at the command line should find the compiler. Version 4.5 of `gfortran` should also work fine.

Installing `gfortran` on OsX using fink/macports:

`gfortran` can also be installed by the package managers but there are some issues with this. `gfortran` is included as part of fink's gcc packages (for example `gcc45`) but it will need to be installed as 64bit, otherwise there will be some compatibility problems with UMFPACK (unless this is modified). With macports it is also part of gcc but needs to be specified as a variant:

```
sudo port install gcc46 +gfortran
```

Again, compilation will take some time. I have not tested this fully.

3.1.5 Pardiso

The pardiso sparse linear matrix solver is included as part of the Intel Math Kernel Library which is packaged with the Intel Fortran compiler (see above).

If using the intel compiler then this solver will automatically become available (check the initial output when running `arb` to see if this has been found).

3.1.6 UMFPACK

The UMFPACK sparse linear solver is part of the suitesparse collection of sparse matrix routines written by Prof. Tim Davis. It is written in c and released under the GNU GPL (see conditions here).

UMFPACK depends on 'METIS - Serial Graph Partitioning and Fill-reducing Matrix Ordering'. METIS is freely available but not free distributable. For more details see the conditions here.

The installation process for the UMFPACK/METIS combination has been automated so that it can be easily used with arb. To install these packages and compile them in a form that is suitable for arb:

```
cd src/contributed/suitesparse
make
```

The `make` command will download the latest version of UMFPACK and version 4.0 of METIS (using curl), and then compile these libraries using the gcc compiler. A wrapper script for using UMFPACK from fortran (included with UMFPACK) will also be compiled. The files will be placed in the src/contributed/suitesparse directory, so will not overwrite any alternative suitesparse or metis libraries already on your system. You need to have `gcc` installed on your system to build these libraries.

If all goes well the following files will be placed in src/contributed/suitesparse to be used by arb:

```
libamd.a
libcamd.a
libccolamd.a
libcholmod.a
libcolamd.a
libmetis.a
libumfpack.a
umf4_f77wrapper.o
```

All of these files are required for UMFPACK to successfully run.

You also need to have a version of the blas libraries available for `arb` to use UMFPACK. On OsX these should already be installed as part of Xcode (see section 3.1.4). On ubuntu linux they can be installed using `sudo apt-get install libblas-dev` if they're not already present.

To remove the compiled libraries and files type

```
make clean
```

from the `suitesparse` directory. This will leave only the downloaded files (ready to be reused). To remove the suitesparse and metis downloads as well, type

```
make clean_all
```

Note that UMFPACK library compilation is dependent on the type of machine architecture and the libraries will need to be remade if transferred from one machine to another - do a `make clean` and then a `make` again.

arb has been tested with the following combinations of umfpack and metis:

- UMFPACK.tar.gz 02-Jun-2010 11:46 and metis-4.0.tar.gz (4.0.1)

3.1.7 Perl

On ubuntu or OsX you should already have a version of perl installed.

3.1.8 Harwell Subroutine Library

An interface to the subroutine MA28D has been implemented. With the availability of UMFPACK, there is no real reason to use this, except for development. For more information see the `src/contributed/hsl_ma28d` directory. An interface to MA48D is under development (don't hold your breath though!).

3.1.9 Numerical Recipes in Fortran 77

These are an alternative to using some of the lapack routines. There is no reason to use these, except for development. For more information see the `src/contributed/numerical_recipes` directory.

3.1.10 Gmsh

While not integral to the arb code, the mesh and data format which arb uses is that developed for gmsh. Gmsh is a mesh element generator which can be run using scripts or via a graphical interface and can be used for post-processing (visualisation) too. Gmsh uses the GNU General Public License (GPL).

There is some great introductory material available on the gmsh website on the use of this program, particularly these online screencasts.

Installing `gmsh` on ubuntu 10.04:

```
sudo apt-get install gmsh
```

On earlier ubuntu versions the repository version is too old.

Installing `gmsh` on anything else:

Download a version from the gmsh website. Version 2.4 is the minimum required for arb - most importantly the msh file format produced needs to be 2.1 or greater. I have had no problems using development versions that are available.

3.2 Installing arb

As arb is distributed as source code that is compiled for each application, arb is not installed in the traditional sense. Instead, for each new simulation a new

self-contained copy of the source files is unpacked from a source tarball. Once created arb is run from a working directory which contains a specific structure of files and subdirectories. Two routines, `pack` and `unpack`, are provided to automate the process of managing this required file/subdirectory structure.

3.2.1 Unpacking the code

To create a new version of arb, a new working directory should be created or downloaded that contains the four files

```
archive.tar.gz
unpack
README
LICENSE
```

Using the command

```
./unpack
```

from within this directory will unpack the archive ready for use.

In the remainder of this section details the file structure and how to pack up the code again.

3.2.2 The working directory and file structure

Once the archive is unpacked the working directory will contain the following subdirectories and files/links:

- `src` directory: contains the main fortran source code of arb, and the `makefile` necessary to build everything (except for the contributed libraries). It also contains the meta-programming perl script `setup_equations.pl` and a template file `equations_module.f90` which are used to create the fortran file `equations_module.f90` within `build` which is specific to each problem.

- src_equations/contributed directory: This directory may/should/can contain contributed third party code that can be used by arb, along with associated interface modules — for example, linear solver routines. There is a separate subdirectory for each package. Each directory contains error handling modules that handle runtime cases where the third party routines are not available, and also some brief installation instructions. Most directories work on the drop-box principle — if the required files are available then they will be included in the arb executable.

- build directory: all building is done within this directory.

- tmp directory: temporary files are stored within this directory, including the file setup/debug which is used for debugging the equation setup.

- output directory: output files from a simulation are placed in here.

- misc directory: contains miscellaneous files — for example a simple script create_mesh which builds a .msh file from any .geo file in the working directory.

- doc directory: contains documentation including this manual.

- examples directory: example problem-specific files equations.in, constants.in and any geometry files (structure files with .geo extension, and mesh files with .msh extension) are stored here. Examples that are detailed in this manual (section 7) are stored in the examples/manual subdirectory.

- packer directory: contains various files specific to the pack and unpack operations. A history file within this directory records the distribution history of the specific code version.

- license directory: contains license details, including the (GNU GPL) license under which arb is released.

- equations.in and constants.in files: these files and an associated .msh or .geo file contain all the problem-specific information required for a particular simulation. equations.in is read by setup_equations when creating the equation-specific fortran code equations_module.f90 within build. constants.in is read by both setup_equations and by the executable fortran code arb. It contains constant and numerical simulation data which can be (mainly) edited without requiring arb to be recompiled.

- arb script: this shell script is a wrapper script for setting-up, making and running arb. You can pass options to this script to control the compilation

process — for example choose between the gnu and intel compilers. It also performs simulation restart file management (not implemented!). This script works out when the equation meta-programming has to be redone or not, and when recompilation is necessary. So if in doubt, just type `arb`.....

- `pack` file: this shell script is used to pack up a simulation.

3.2.3 Packing the code

To pack a simulation ready to transport or backup, use the command

```
./pack
```

from within the working directory. This will create a subdirectory with a name of the form `arb_[date]_[version]` which contains all files necessary to run arb. Following the command with a name, as in

```
./pack a_name
```

will create the archive in a subdirectory named `a_name` instead of the default.

The script `pack` accepts a number of options. By default only files within the `examples` or `gmsh` directories that are specific to this manual are included in the archive. The options `--example`, `--misc` or `--all` specify that all files within either the `example`, `misc` or both directories are contained within the archive. By default only source code within the `src/contributed` directory that is not subject to a non-free third party license is included in the archive. Using the options `--contributed` or `--all` causes all files in these directories to be archived (including the build suitesparse libraries). Using `--distribute` means that no third party software is included in the archive. The option `--build` means that all files in the build directory will be included in the archive. This may be useful if you want to transport the simulation to another machine that may not have maxima installed (for example).

Other options to `pack` include `--notar` and `--help`.

4 Running simulations

4.1 A super-quick example: A heat conduction simulation

Once the code is unpacked you should be left in the working directory. If this version has been downloaded from the homepage then the three files

```
constants.in
equations.in
surface.msh
```

that exist in the working directory specify a simple heat conduction problem. If they don't exist for whatever reason they can be copied from `examples/manual/heat_conduction_ar` to the working directory.

Check that the file `constants.in` specifies your choice of linear solver. If you are using the gnu compiler with the UMFPACK routines then this file should already contain the line

```
LINEAR_SOLVER "SUITESPARSE_UMF" # suitesparse umf solver
    (UMFPACK) by Timothy A. Davis
```

so you don't need to change anything. If you are using the intel compiler with MKL libraries then change this line to

```
LINEAR_SOLVER "INTEL_PARDISO" # pardiso solver contained
    in intel mkl library
```

Now run the simulation using

```
./arb
```

If all goes well the code will be metaprogrammed and compiled, the simulation will run and output will be produced in the directory `output`. To view the output type

```
gmsh output/output.msh
```

You should see the temperature field around a heated ellipse.

4.2 A more detailed guide: Newtonian fluid flow

The information about running a simulation here is presently out of date, but there is some useful info about the mesh and input file structure.

To discuss the working method in more detail we use the example of steady-state flow of a Newtonian fluid around a cylinder that is contained in a 2d channel.

There are 5 basic steps to setting up and running an arb simulation:

1. **Create a mesh:** Geometry definition and mesh creation can be performed in gmsh. The domain geometry details are stored in a `.geo` file, which can be created by hand (file editing) or with the help of the gmsh GUI. An example `.geo` file is included in Section 7.1 for the considered case of the 2d channel containing a cylinder. This file is included in the `gmsh/manual/2d_channel_with_cylinder/` directory.

   Once a geometry (`.geo` file) has been created, gmsh can mesh this to produce a `.msh` file which is the file that is read in by arb. For the considered test case place the generated mesh file `surface.msh` in the `gmsh/manual/2d_channel_with_cylinder/` directory. When creating a 3D mesh be sure to optimise the mesh after creation (greatly improves mesh quality).

   arb uses the concept of 'regions' to locate various equations (for example boundary conditions, domain equations etc) and these must be defined in the `.geo` file prior to meshing. Regions have names that are delimited by the < and > signs: for example `<inlet>` and `<outlet>` (generally any user-defined names are delimited this way in arb).

   In the example geometry file the boundary regions `<inlet>`, `<outlet>` and `<cylinder>` have all been defined as physical entities. The area region of `<flow domain>` which contains all mesh cells within the flow domain has also been created. This region is necessary as by default, gmsh does not write out mesh cells to a mesh file unless they are part of a physical entity. Note that certain region names are reserved: see section 5.3 for more details.

2. **Edit** `equations.in` **and** `constants.in`: Aside from the mesh file, all information that is specific to a simulation is contained in these two `.in` files. `equations.in` is read only by the `setup_equations` perl script when performing the fortran 'meta-programming'. `constants.in` is also read by this perl script, but is also read by the `arb` fortran executable each time a

simulation is run. In general physical constants that would change regularly between similar simulations should be defined in `constants.in` as their values can be changed without having to rerun the meta-programming script `setup_equations`. On the other hand all equations, which are processed by maxima via `setup_equations`, must be defined in `equations.in`. Hence any change to `equations.in` requires `setup_equations` to be rerun and the fortran executable `arb` remade.

Note that some simulation specific information such as the mesh file name, newton solver convergence tolerance, number of newton iteration steps and dimensions of the physical problem are defined in the `constants.in` file and read directly by the `arb` executable: Hence these can be changed without reruning `setup_equations` or recompiling `arb`. Similarly any region definitions contained in `constants.in` are only read by the `arb` exectuable and can be changed without rerunning `setup_equations`.

Section 7.1 gives example `equations.in` and `constants.in` files for the considered test case of steady-state fluid flow around a cylinder. These files are given in `examples/manual/steady_state_channel_flow_with_cylinder` and should be copied to the working directory. The quoted string after the `READ_GMSH` keyword in `constants.in` should refer to the location of the mesh file, relative to the working directory.

Further details regarding the syntax of both `equations.in` and `constants.in` can be found in Sections 5 and 6.

3. **Run `setup_equations`:** From the working directory running

   ```
   ./setup_equations
   ```

   reads both `equations.in` and `constants.in`, and using maxima, creates the fortran source code file `src/equations_module.f90`. `setup_equations` writes some progress information to the screen - if all goes well this output will end with the statement '`success`'.

   If the script is not successful then errors in the `equations.in` and `constants.in` files will need to be found and corrected. Aside from the screen output, more debugging information is written by `setup_equations` to the file `tmp/debug`. Other files in the `tmp` directory trace the interaction between the perl script and maxima.

4. **Compile and run arb:** The `arb` script in the working directory is a 'wrapper' for handling the running of arb. Typing

   ```
   ./arb -q -m &
   ```

from the working directory will remove old output files, compile `arb` (make option `-m`), run `arb` and direct the output to `output/output.scr` (quiet option `-q`), all in the background (`&`).

Other options to the `arb` run script include `-d` to make with debugging options and run in a debugging environment, and `-c` to continue from a previous run (not implemented yet).

5. **View results:** arb produces a file `output/output.msh` which can be opened by gmsh for viewing. This file includes the mesh information as well as variable data, so can be passed to arb as an input `.msh` file for subsequent simulations (although only mesh read, not data read, is currently implemented).

5 Inside arb

5.1 Code structure

5.2 Mesh structure

arb uses an unstructured mesh composed of cell elements that are separated by face elements. The dimension of the domain cell elements is specified in `constants.in` using the `DIMENSIONS` keyword. The dimension of the face elements is always 1 less than that of the domain cell elements. arb has been coded to be able to handle any poly-sided cells, however in practice it has only been tested to date (v0.23) with tetrahedron in 3D, triangles in 2D and lines in 1D: these are the default element geometries created by gmsh.

Boundary cells are created by arb after a mesh has been imported. They have a dimension that is 1 less than that of the domain cells, so have no volume/area/width in 3D/2D/1D, respectively. Each boundary cell has the same geometry, and is conincident with, a boundary face. Hence, a mesh has the same number of boundary faces as boundary cells.

5.3 Regions

Regions are sets of mesh elements that are used to locate user-defined variables and equations. Each region may contain only mesh elements of the same centring (that is, either cell or face elements, but not both). Regions can be defined by the user directly in gmsh when the mesh is generated, or via statements in the `constants.in` file that are interpreted when `arb` is run. There are also several generic system generated regions. Region names must be delimited by the <> characters, but apart from these two characters their names may contain any non-alphanumeric characters.

5.3.1 Defining regions via gmsh

Regions are specified in gmsh by defining and then naming physical entities. To do this via the gmsh GUI:

- Add a physical entity (under the physical groups tab) by selecting various elemental entities.

- Edit the geometry file (using the edit tab) and change the physical entity's name from the numerical name given by gmsh to the required <> delimited name suitable for arb.
- Save the `.geo` file.
- Reload the `.geo` file again (using the reload tab). If you now check under the visibility menu the physical entity will be visible.

5.3.2 Defining regions within the `constants.in` file

Currently there are two types of region specification statements that can be used in the `constants.in` file:

Compound region:

```
CELL_REGION <name> "COMPOUND +<region1>+<region2>-<
   region3>" # comments
FACE_REGION <name> "COMPOUND <region1>-<region2>" #
   comments
```

A compound region is defined using other existing regions. All regions that are used in the definition (ie, `<region1>`, `<region2>` and `<region3>` in the above examples) must have the (same) centring that is specified by the `REGION` keyword. If a + sign precedes a region name in the list of regions, then all the mesh elements that are in the following region are added to the new compound region, if they are not already members. If a – sign precedes a region name in the list of regions, then all the mesh elements that are in the following region are removed from the new compound region, if they are (at that stage) members of the new compound region. If no sign immediately precedes a region name in the defining list then a + sign is assumed. When constructing a compound region `arb` deals with each region in the defining list sequentially; so whether a mesh element is included in the compound region or not may depend on the order that the regions are listed.

At region:

```
CELL_REGION <name> "AT x1 x2 x3" # comments
FACE_REGION <name> "AT x1 x2 x3" # comments
```

This statement defines a region that contains one cell or one face mesh element. The element chosen lies closest to the point `(x1,x2,x3)`. The values `x1`, `x2` and `x3` can be real or double precision floats.

5.3.3 System generated regions

The following regions are generated by `arb` at the start of a simulation. The names cannot be used for user-defined regions:

region name	description
`<all cells>`	all cells
`<domain>`	internal domain cells
`<boundary cells>`	cells located on the boundary
`<all faces>`	all faces
`<domain faces>`	internal domain faces
`<boundaries>`	faces located on the boundary

Additionally, there are a number of system regions which may be used in user-written expressions (see section 6) which specify sets of mesh elements relative to the current position. These names cannot be used for user-defined regions either:

region name	rel. to	description
`<celljfaces>`	cell	faces that surround the current cell
`<nobcelljfaces>`	cell	faces that surround the current cell, unless the current cell is on a boundary. In that instance move to the neighbouring domain cell and then cycle around the surrounding face cells.
`<cellicells>`	cell	cells that are local to the current cell (more than just the adjacent cells)
`<faceicells>`	face	cells that are local to the current face (more than just the adjacent cells)
`<adjacentcellicells>`	cell	cells that are strictly adjacent to the current cell
`<adjacentfaceicells>`	face	cells that are strictly adjacent to the current face (always two)

<upwindfaceicells>	face	the cell that is upwind of the face, used when performing faceave[advection] averaging (see section 6. Not really a user region.
<downwindfaceicells>	face	the cell that is downwind of the face, used when performing faceave[advection] averaging (see section 6. Not really a user region.
<cellkernel[l=0]>	cell	surrounding faces used in a cell averaging kernel (see section 6. Not really a user region.
<cellkernel[l=1-3]>	cell	surrounding cells used in cell derivative kernels (see section 6. Not really a user region.
<cellkernel[l=4]>	cell	surrounding nodes used in a cell averaging kernels (see section 6. Not really a user region.
<facekernel[l=0-6]>	face	surrounding cells used in face averaging and derivative kernels (see section 6. Not really a user region.
<noloop>	face/cell	dummy region which specifies no elements.

5.4 Variable types

There are six types of user defined variables: constant, transient, derived, unknown, equation and output. Each of these are stored in arb using the same general data structure (fortran type var). Any of these variables can be defined by a user-written expression in equations.in which is read by setup_equations and interpreted by maxima. Additionally, the constant type may be defined in constants.in and there given (only) a numerical value. Along with the user defined variables, there are also system defined variables which can be used in user-written expressions.

All variables have an associated compound variable type (scalar, vector or tensor) which is used mainly for output purposes.

Details of both the user and system defined variables are given in this section.

5.4.1 Constant type variable defined in equations.in

Synopsis:

Constant variables are evaluated once at the start of a simulation. If defined in equations.in they are defined using an expression which may contain only system variables and other constants — in the latter case the constants must

have been defined in either the `constants.in` file or previously (above) in the `equations.in` file.

Defining statements:

```
CELL_CONSTANT <name> [multiplier*units] "expression" ON
   <region> options # comments
FACE_CONSTANT <name> [multiplier*units] "expression" ON
   <region> options # comments
NONE_CONSTANT <name> [multiplier*units] "expression"
   options # comments
CONSTANT <name> [multiplier*units] "expression" options
   # comments
```

Statement components:

- `(CELL_|FACE_|NONE_|)CONSTANT` *(required)*: This keyword specifies the centring of the variable. Constants that have cell or face centring vary over the simulation domain, and have values associated with each cell or face, respectively (subject to the `region` statement, below). None centred constants have one value that is not linked to any spatial location. If the centring specifier is omitted from the keyword (as in `CONSTANT`) then none centring is assumed (ie., keyword `CONSTANT` is equivalent to keyword `NONE_CONSTANT`).

- `<name>` *(required)*: Each variable must have a unique name, delimited by the < and > characters. Besides these characters, the variable may contain spaces and any other non-alphanumeric characters. If the name ends with a direction index, as in `<u[l=1]>` or `<gradp [l=3]>`, then the variable is considered to be a component of a three dimensional vector compound. Similarly, if the name ends with a double direction index, as in `<tau[l=1,3]>`, the variable is considered to be a component of a three by three tensor compound. Components of compounds that are not explicitly defined are given a zero value (when used in dot and double dot products for example). All defined components that are members of the same compound must be of the same variable type, have the same centring, be defined over the same region and have the same units and multiplier. Certain names are reserved for system variables (see section 5.4.9).

- `multiplier` *(optional)*: When reading in numerical constants, each value is multiplied by this value. At present not in use in `equations.in`.

- `units` *(optional):* A string which specifies the units for the variable. At present this string is not interpreted by the code at all and the user must ensure that the units used are consistent.

- `"expression"` *(required)*: When a constant is defined in `equations.in`, this double-quoted expression is used to specify the value of the constant. As they may contain system variables and also other constants, they may vary throughout the domain. For more details regarding the syntax of these expressions, see section 6.

- `ON <region>` *(optional)*: This part of the statement determines over what region the variable should be defined. It is only applicable for cell and face centred variables, and must in these cases refer to a region that has the same centring as the variable. If ommitted then by default a cell centred constant will be defined on `<all cells>` and a face centred constant on `<all faces>`. Note that refering to a variable value outside of its region of definition will produce an error when running `arb`.

- `options` *(optional)*: This is a comma separated list of options. Valid options for the constant variable type include:

  - `componentoutput`: This component to be written to `output.msh`. Default is `nocomponentoutput` for constants.
  - `compoundoutput`: The compound variable that is component is a member of to be written to `output.msh`. If the compound is cell centred then the data is output at cell and node centres (which looks better than only cell centred data when rendered). Default is `nocompoundoutput` for constants.

- `comments` *(optional)*: Anything written beyond the # is regarded as a comment.

Examples:

```
CELL_CONSTANT <test constant> "<cellx[l=1]>^2" ON <
   boundaries> # a test
FACE_CONSTANT <test constant 2> [m] "<facex[l=2]>" #
   another test
```

5.4.2 Constant type variable defined in `constants.in`

Synopsis:

Constant variables defined in `constants.in` are set to numerical values read directly by the `arb` executable, rather than expressions interpreted by maxima.

Defining statements:

```
CELL_CONSTANT <name> [multiplier*units] value ON <region
   > options # comments
FACE_CONSTANT <name> [multiplier*units] value ON <region
   > options # comments
NONE_CONSTANT <name> [multiplier*units] value options #
   comments
CONSTANT <name> [multiplier*units] value options #
   comments
```

Statement components:

The components of these statements are the same as in section 5.4.1 with the exception of:

- `value` *(required)*: A numerical value of real or double precision type.

Examples:

```
CONSTANT <mu> [Pa.s] 1.0d-3 # fluid viscosity
NONE_CONSTANT <rho> [997*kg/m^3] 1.0 # fluid density
```

5.4.3 Constant type variable defined per region in `constants.in`

Synopsis:

This definition can be used in the `constants.in` file to assign different numerical values to either a cell or face centred constant in specific regions. Two statements are required for this type of constant definition: The first defines the list of regions where the next constant will be set (`REGION_LIST`) and the second defines the constant and sets/lists the corresponding numerical values (`(CELL_|FACE_)REGION_CONSTANT`). The region names in the `REGION_LIST`

statement must have the same centring as the following REGION_CONSTANT statement. Furthermore, the <region> over which the constant is defined must include all of the regions listed within the previous REGION_LIST statement.

Defining statements:

```
REGION_LIST <region1> <region2> ... <regionN> # comments
CELL_REGION_CONSTANT <name> [multiplier*units] value1
   value2 ... valueN ON <region> options # comments
FACE_REGION_CONSTANT <name> [multiplier*units] value1
   value2 ... valueN ON <region> options # comments
```

Statement components:

The components of these statements are the same as in section 5.4.1 with the exception of:

- <region1> <region2> ... <regionN> *(required)*: A list of regions that have the same centring as the following REGION_CONSTANT statement.
- value1 value2 ... valueN *(required)*: A list of numerical values for the constant, corresponding in a one-to-one fashion with the list of regions given in the previous REGION_LIST statement.

Examples:

```
REGION_LIST <inlet> <outlet> # some face regions
FACE_REGION_CONSTANT <electric field> [V/m] 10 20. ON <
   boundaries>
```

5.4.4 Transient type variable defined in `equations.in`

Synopsis:

These will be implemented for version 0.3 (transient).

5.4.5 Derived type variable defined in `equations.in`

Synopsis:

Derived variables depend on the unknown variables and other previously defined (ie, above in the file) derived variables.

Defining statements:

```
CELL_DERIVED <name> [multiplier*units] "expression" ON <
   region> options # comments
FACE_DERIVED <name> [multiplier*units] "expression" ON <
   region> options # comments
NONE_DERIVED <name> [multiplier*units] "expression"
   options # comments
DERIVED <name> [multiplier*units] "expression" options #
    comments
```

Statement components:

Along with the information presented in section 5.4.1, the following applies to derived variables:

- `(CELL_|FACE_|NONE_|)DERIVED` *(required)*: If no centring is specified then none centring is assumed.
- `"expression"` *(required)*: This is an expression for the derived variable in terms of constant, transient, unknown, previously defined derived (appearing above in `equations.in`) and system variables.
- `ON <region>` *(optional)*: If ommitted then by default a cell centred derived will be defined on `<all cells>` and a face centred derived on `<all faces>`.
- `options` *(optional)*: This is a comma separated list of options. Valid options for derived variables (as well as those given in section 5.4.1) include:
  - `noderivative`: Normally the derivative of this variable's expression is calculated with respect to each unknown variable (the Jacobian) when performing the Newton-Raphson solution procedure. Including this option sets this derivative to zero. This may be required for functions for which the derivative cannot be calculated or for functions that undergo step changes (not continuous) which are not ammeniable to solution via the Newton-Raphson procedure. Using this option will usually slow convergence.

– `positive/negative`: Including one of these options causes the code to check the sign of the derived variable. This is particularly useful for quantities like concentrations that are only physically meaningful when being positive. By using an expression such as `"1-<con>"` and including the option `positive` an upper limit for a variable can also be enforced.

Examples:

```
FACE_DERIVED <tau[l=1,1]> "<p> - <mu>*2.d0*facegrad[l
   =1](<u[l=1]>)" compoundoutput
CELL_DERIVED <graddivp[l=1]> "celldivgrad[l=1](<p>)" #
   divergence based pressure gradient
```

5.4.6 Unknown type variable defined in `equations.in`

Synopsis:

Unknown variables are those upon which the equations and derived variables ultimately depend.

Defining statements:

```
CELL_UNKNOWN <name> [multiplier*units] magnitude "
   expression" ON <region> options # comments
FACE_UNKNOWN <name> [multiplier*units] magnitude "
   expression" ON <region> options # comments
NONE_UNKNOWN <name> [multiplier*units] magnitude "
   expression" options # comments
UNKNOWN <name> [multiplier*units] magnitude "expression"
    ON <region> options # comments
```

Statement components:

Along with the information presented in section 5.4.5, the following applies to unknown variables:

- `(CELL_|FACE_|NONE_|)UNKNOWN` *(required)*: If no centring is specified then cell centring is assumed.

- `magnitude` *(required)*: An order of magnitude estimate (postive and greater than zero real or double precision value) must be specified for all unknown variables. This magnitude is used when checking on the convergence of the solution.
- `"expression"` *(required)*: For an unknown variable the expression specifies the variable's initial value. The expression may contain constants, previously defined (initial) derived values (those appearing above in `equations.in`) and system variables.
- `ON <region>` *(optional)*: If ommitted then by default a cell centred unknown will be defined on `<all cells>` and a face centred unknown on `<all faces>`.
- `options` *(optional)*: The `noderivative` option is not applicable for unknown variables.

Examples:

```
CELL_UNKNOWN <u[1=1]> 1.d0 "<u_av>" # a velocity
   component
CELL_UNKNOWN <p> [] 1.d0 "1.d0-<cellx[1=1]>" # pressure
NONE_UNKNOWN <p_in> [Pa] 1.d0 "1.d0" # the pressure at
   the inlet
```

5.4.7 Equation type variable defined in `equations.in`

Synopsis:

Equation variables represent the equations to be satisfied. The equation expressions should be formulated so that when the equation is satisfied, the expression equals zero. The number of equations must equal the number of unknown variables. Furthermore, for the system to be well posed the equations must be unknown (no single equation can be made from a combination of the other equations).

Defining statements:

```
CELL_EQUATION <name> [multiplier*units] "expression" ON
   <region> options # comments
```

```
FACE_EQUATION <name> [multiplier*units] "expression" ON
   <region> options # comments
NONE_EQUATION <name> [multiplier*units] "expression"
   options # comments
EQUATION <name> [multiplier*units] "expression" options
   # comments
```

Statement components:

Along with the information presented in section 5.4.5, the following applies to equation variables:

- `(CELL_|FACE_|NONE_|)EQUATION` *(required)*: If no centring is specified then none centring is assumed.
- `"expression"` *(required)*: For an equation variable the expression should equal zero when the equation is satisfied. The expression may contain constant, transient, derived, unknown and system variables.
- `ON <region>` *(optional)*: If ommitted then by default a cell centred equation will be defined on `<domain>` and a face centred equation on `<boundaries>`.

Examples:

```
CELL_EQUATION <continuity> "celldiv(<u_f>)" ON <domain>
   # continuity
FACE_EQUATION <outlet noslip> "dot(<u[l=:]>,<facetang1[l
   =:]>)" ON <outlet> # no component tangential to
   outlet
NONE_EQUATION <p_in for flowrate> "<u_av_calc>-<u_av>" #
    set flowrate through inlet to give required average
   velocity
```

5.4.8 Output type variable defined in `equations.in`

Synopsis:

Output variables are evaluated once convergence of the solution has been reached: They are only for output purposes.

Defining statements:

```
CELL_OUTPUT <name> [multiplier*units] "expression" ON <
   region> options # comments
FACE_OUTPUT <name> [multiplier*units] "expression" ON <
   region> options # comments
NONE_OUTPUT <name> [multiplier*units] "expression"
   options # comments
OUTPUT <name> [multiplier*units] "expression" options #
   comments
```

Statement components:

Along with the information presented in section 5.4.5, the following applies to output variables:

- `(CELL_|FACE_|NONE_|)OUTPUT` *(required)*: If no centring is specified then none centring is assumed.
- `"expression"` *(required)*: For an output variable the expression may contain constant, transient, derived, unknown, equation and system variables.
- `ON <region>` *(optional)*: If ommitted then by default a cell centred output variable will be defined on `<all cells>` and a face centred output variable on `<all faces>`.
- `options` *(optional)*: The `noderivative` option is not applicable for output variables (this option is implicitly set anyway for these variables).

Examples:

```
NONE_OUTPUT <F_drag> [N] "facesum(<facearea>*dot(<
   facenorm[1=:]>,<tau[1=:,1]>),<cylinder>)" # force on
   object in axial direction
```

5.4.9 System variables

5.5 Simulation options

5.6 Data visualisation

6 Expression language reference

Sorry - there's lots missing in this section!

6.1 Operators

6.1.1 `celldiv`: Divergence calculated around a cell

Summary: Uses Gauss' theorem to calculate the divergence of a face centred component around a cell.

Statement:

`celldiv[options](face_centred_expression)`

Centring:

Operator is cell centred, contents of operator is face centred.

Details:

Using Gauss' theorem to evaluate divergences around cells is probably the defining characteristic of Finite Volume methods. `celldiv` performs this operation.

To discretise the divergence of a vector $\boldsymbol{u}$ over a cell within the domain, Gauss' theorem gives

$$\frac{\int_{V_{\text{cell}}} \boldsymbol{\nabla} \cdot \boldsymbol{u} dV}{\int_{V_{\text{cell}}} dV} = \frac{1}{V_{\text{cell}}} \int_{S_{\text{cell}}} \boldsymbol{n}_{\text{cell}} \cdot \boldsymbol{u} dS = \frac{1}{V_{\text{cell}}} \sum_j (\boldsymbol{n}_{\text{cell}} \cdot \boldsymbol{n}_j)(\boldsymbol{u} \cdot \boldsymbol{n}_j) S_j$$

where V_{cell} and S_{cell} are the volume and total surface area of the cell, respectively, $\boldsymbol{n}_{\text{cell}}$ is a unit normal pointing outward from the cell, $\boldsymbol{n}_j$ is a normal associated with surrounding face j, and the sum is conducted over all faces (index j) which surround the cell. Taking the divergence of a vector results in a scalar. The above vector divergence is represented by `celldiv` as

```
celldiv(dot(<u[1=:]>,<facenorm[1=:]>))
```

where in this case the vector $\boldsymbol{u}$ is represented by the three component variables `<u[1=1]>`, `<u[1=2]>` and `<u[1=3]>`, and the unit normal associated with

the face j, $\boldsymbol{n}_j$, is given by the system component variables <facenorm[l=1]>, <facenorm[l=2]> and <facenorm[l=3]>.

Note that the region used by arb in performing the sum $\sum_j$ is <nobcelljfaces>. This relative region specifies all faces that surround a given cell, unless that cell is a boundary cell. As boundary cells are not fully surrounded by faces Gauss' theorem can not be applied. Hence, if the operator celldiv is used at a boundary cell then the region <nobcelljfaces> is taken relative (moved) to the closest domain cell that is adjacent the boundary cell, so this is where celldiv becomes evaluated. Physically it is inadvisable to use an equation that involves a divergence at a boundary cell anyway.

Options:

- noderivative: No derivatives with respect to the unknown variables for the Newton-Raphson Jacobian are calculated for this operator (and its contents).

Examples:

```
CELL_EQUATION <continuity> "celldiv(<u_f>)" ON <domain>
   # continuity equation
CELL_EQUATION <momentum[l=1]> "celldiv(<J_f[l=1]>)" ON <
   domain> # momentum conservation in direction l=1
CELL_EQUATION <momentum[l=2]> "celldiv(<J_f[l=2]>)" ON <
   domain> # momentum conservation in direction l=2
```

6.1.2 `cellgrad` or `facegrad`: Gradient evaluated at a cell or face

6.1.3 `celldivgrad`: Gradient evaluated at a cell calculated via a divergence

6.1.4 `cellave` or `faceave`: Interpolation of a quantity from one centring to another

6.1.5 `celllimiter`: Gradient limiter for ensuring advection stability

6.1.6 `cellif` or `faceif` or `noneif`: If conditional statement

6.1.7 `cellsum` or `facesum`: Sum performed over a region of elements

6.1.8 `cellmax` or `facemax` or `nonemax`: Picks the maximum from a region of elements

6.1.9 `cellmin` or `facemin` or `nonemin`: Picks the minimum from a region of elements

6.1.10 `celldelta` or `facedelta`: A delta function to identify specific regions

7 Example applications

7.1 Incompressible steady-state Newtonian flow through a 2D channel containing a cylinder with set inlet velocities

Listing 1: ../../examples/manual/steady_state_channel_flow_with_cylinder/surface.geo

```
// 2d_channel_with_cylinder

lc = 0.05;  // charateristic mesh length variable

// setup domain boundaries
Point(1) = {0, 0, 0, lc/2};
Point(15) = {0.2, 0, 0, lc/4};
Point(2) = {2.2, 0,  0, lc} ;
Point(3) = {2.2, 0.41, 0, lc} ;
Point(16) = {0.2, 0.41, 0, lc/4};
Point(4) = {0,  0.41, 0, lc/2} ;

Line(1) = {1,15} ;
Line(15) = {15,2} ;
Line(2) = {2,3} ;
Line(3) = {3,16} ;
Line(16) = {16,4} ;
Line(4) = {4,1} ;

// create an elementary entity that is the domain
   boundary
Line Loop(5) = {1,15,2,3,16,4} ;

// create the physical entities for the inlet and output
    which become the arb regions
Physical Line("<inlet>") = {4};
Physical Line("<outlet>") = {2};

// create the cylinder
Point(5) = {0.2, 0.15, 0, lc/4};
Point(6) = {0.25, 0.2, 0, lc/4};
Point(7) = {0.2, 0.25, 0, lc/4};
Point(8) = {0.15, 0.2, 0, lc/4};
Point(9) = {0.2, 0.2, 0, lc/4};
```

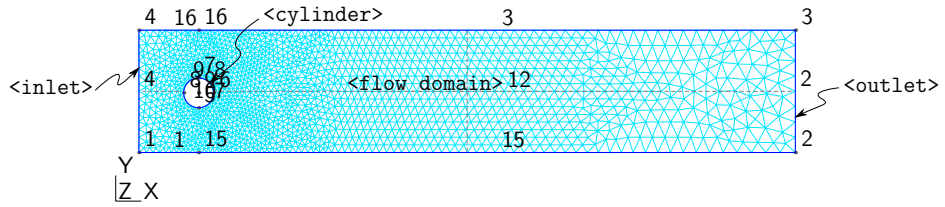

Figure 1: Geometry and resulting mesh from the surface.geo file for the flow through a 2d channel containing a cylinder problem.

```
Ellipse(7) = {5, 9, 9, 6};
Ellipse(8) = {6, 9, 9, 7};
Ellipse(9) = {7, 9, 9, 8};
Ellipse(10) = {8, 9, 9, 5};

// create an elementary entity that is the cylinder
   boundary
Line Loop(11) = {7, 8, 9, 10};

// create the physical entity for the cylinder boundary
   which becomes the arb region
Physical Line("<cylinder>") = {7, 8, 9, 10};

// all of the flow domain must be included as a physical
    entity to be output under gmsh
Plane Surface(12) = {5, 11};
Physical Surface("<flow domain>") = {12};
```

Listing 2: ../../examples/manual/steady_state_channel_flow_with_cylinder/constants.in

```
# arb finite volume solver
# Copyright 2009,2010 Dalton Harvie (daltonh@unimelb.edu
   .au)
#
# arb is released under the GNU GPL.  For full details
   see the license directory.
#
#
   ------------------------------------------------------------

# file constants.in
VERSION 0.25
```

```
#
   ------------------------------------------------------------

# user-defined constants

# REGION_LIST line has the ordered names of any regions
   (quoted)
# (CELL_|FACE_)REGION_CONSTANT sets a constant that
   varies with region and has the form: <name> [
   multiplier*units] value_for_region_1 ..
   value_for_region_n options # comments
# (CELL_|FACE_|NONE_|)CONSTANT sets a constant specific
   to one region or no regions and has the form: <name>
   [multiplier*units] value ON <region> options #
   comments

# physical data
CONSTANT <mu> [Pa.s] 1.d-3 # viscosity of liquid
CONSTANT <rho> [kg/m^3] 1.d0 # density
CONSTANT <u_av> [m/s] 0.2d0 # average inlet velocity

# numerical data
CONSTANT <C_{Rhie-Chow}> [] 1.0d+0 # multiplier for Rhie
   -Chow-type pressure oscillation control
CONSTANT <adv_limiter> [] 1.d0 # multiplier used to
   limit gradients when calculating advection fluxes

#
   ------------------------------------------------------------

# system constants

NEWTRESTOL 1.d-12 # convergence criterion for newton
   solver
NEWTSTEPMAX 20 # maximum number of steps for newton
   solver

#
   ------------------------------------------------------------

# geometry

# CELL_REGION/FACE_REGION specified by: <name> "location
```

```
    string" # comments
# where location string could be: "AT x1 x2 x3" for a
   single point closest to these coordinates
# where location string could be: "COMPOUND +<a region
   >-<another region>" for a + and - compound region
   list

FACE_REGION <walls> "COMPOUND <boundaries>-<inlet>-<
   outlet>"

# DIMENSIONS is the number of dimensions used in the
   problem
DIMENSIONS 2

# READ_GMSH instructs arb to read a gmsh file
READ_GMSH "surface.msh"

# linear solver that is used to invert jacobian
#LINEAR_SOLVER "HSL_MA28" # hsl archive direct solver
#LINEAR_SOLVER "INTEL_PARDISO" # pardiso solver
   contained in intel mkl library
#LINEAR_SOLVER "INTEL_PARDISO_OOC" # pardiso solver
   contained in intel mkl library
LINEAR_SOLVER "SUITESPARSE_UMF" # suitesparse umf solver
    (UMFPACK) by Timothy A. Davis

#
   ------------------------------------------------------------
```

Listing 3: ../../examples/manual/steady_state_channel_flow_with_cylinder/equations.in

```
# arb finite volume solver
# Copyright 2009,2010 Dalton Harvie (daltonh@unimelb.edu
   .au)
#
# arb is released under the GNU GPL.  For full details
   see the license directory.
#
#
   ------------------------------------------------------------

# file equations.in
```

```
VERSION 0.25
#
   ------------------------------------------------------------------------


# statement reference
# (CELL_|FACE_|NONE|)CONSTANT <name> [units] "expression
    (involving only constants)" ON <region> options #
   comments
# (CELL_|FACE_|NONE|)TRANSIENT <name> [units] magnitude
   "expression" ON <region> options # comments
# (CELL_|FACE_|NONE|)DERIVED <name> [units] magnitude "
   expression" ON <region> options # comments
# (CELL_|FACE_|NONE|)UNKNOWN <name> [units] "expression
   (initial value)" ON <region> options # comments
# (CELL_|FACE_|NONE|)EQUATION <name> [units] "expression
    (equation equaling zero)" ON <region> options #
   comments
# (CELL_|FACE_|NONE|)OUTPUT <name> [units] "expression"
   ON <region> options # comments

# options include:
#  noderivative - for DERIVED, EQUATION
#  positive,negative - for DERIVED, UNKNOWN, EQUATION
#  harmonic - for CONSTANT, TRANSIENT, DERIVED, UNKNOWN
#  compoundoutput/nocompoundoutput - for ALL
#  componentoutput/nocomponentoutput - for ALL

# unknown variables used for flow problems
CELL_UNKNOWN <u[l=1]> [] 1.d0 "<u_av>" # velocity
   component
CELL_UNKNOWN <u[l=2]> [] 1.d0 "0.d0" # velocity
   component
CELL_UNKNOWN <p> [] 1.d0 "1.d0-<cellx[l=1]>" # pressure

# total stress tensor
FACE_DERIVED <tau[l=1,1]> "<p> - <mu>*2.d0*facegrad[l
   =1](<u[l=1]>)" compoundoutput
FACE_DERIVED <tau[l=1,2]> "- <mu>*(facegrad[l=2](<u[l
   =1]>)+facegrad[l=1](<u[l=2]>))"
FACE_DERIVED <tau[l=2,2]> "<p> - <mu>*2.d0*facegrad[l
   =2](<u[l=2]>)"
FACE_DERIVED <tau[l=2,1]> "<tau[l=1,2]>"
```

```
# a Rhie-Chow-type correction is applied to the face
   velocities
CELL_DERIVED <graddivp[l=1]> "celldivgrad[l=1](<p>)" #
   pressure gradient calculated via a divergence (
   consistent with momentum conservation)
CELL_DERIVED <graddivp[l=2]> "celldivgrad[l=2](<p>)" #
   pressure gradient calculated via a divergence (
   consistent with momentum conservation)
FACE_DERIVED <p_error> "facegrad(<p>) - dot(<graddivp[l
   =:]>,<facenorm[l=:]>)" # difference between face
   centred and cell divergence type gradient at face and
    normal to the face
FACE_DERIVED <u_f_{correction}> "-<C_{Rhie-Chow}>*
   facedelta(<domain faces>)*facemin(<facedx>^2/<mu>,
   sqrt(<facedx>/(<rho>*facemax(abs(<p_error>),1.d-6)))
   )*<p_error>" compoundoutput # the Rhie-Chow type
   velocity correction, only applied (nonzero) on the
   domain faces

# flux of mass (volume) and momentum components over
   each face
FACE_DERIVED <u_f> "dot(<u[l=:]>,<facenorm[l=:]>) + <
   u_f_{correction}>" # volume (velocity) transport
FACE_DERIVED <J_f[l=1]> "dot(<facenorm[l=:]>,<tau[l
   =:,1]>)+<rho>*faceave[advection](<u[l=1]>,<u_f>,<
   adv_limiter>)*<u_f>" compoundoutput # component of
   momentum transport from stress and advection
FACE_DERIVED <J_f[l=2]> "dot(<facenorm[l=:]>,<tau[l
   =:,2]>)+<rho>*faceave[advection](<u[l=2]>,<u_f>,<
   adv_limiter>)*<u_f>" # component of momentum
   transport from stress and advection

# conservation equations solved over each domain cell (
   finite volume method)
CELL_EQUATION <continuity> "celldiv(<u_f>)" ON <domain>
   # continuity
CELL_EQUATION <momentum[l=1]> "celldiv(<J_f[l=1]>)" ON <
   domain> # momentum component
CELL_EQUATION <momentum[l=2]> "celldiv(<J_f[l=2]>)" ON <
   domain> # momentum component

# boundary conditions on nonslip walls
```

```
FACE_EQUATION <wall noflux> "<u_f>" ON <walls> # no flux
    normal to walls
FACE_EQUATION <wall noslip> "dot(<u[l=:]>,<facetang1[l
   =:]>)" ON <walls> # nonslip
FACE_EQUATION <wall p extrapolation> "dot(<graddivp[l
   =:]>,<facenorm[l=:]>)" ON <walls> # extrapolate
   pressure to the wall using zero gradient normal to
   wall

# boundary conditions on outlet - fully developed flow
FACE_EQUATION <outlet fully developed> "facegrad(dot(<u[
   l=:]>,cellave[lastface](<facenorm[l=:]>)))" ON <
   outlet> # normal velocity component is fully
   developed
FACE_EQUATION <outlet noslip> "dot(<u[l=:]>,<facetang1[l
   =:]>)" ON <outlet> # no component tangential to
   outlet
FACE_EQUATION <outlet p> "<p>" ON <outlet> # specified
   uniform (zero) pressure

# boundary conditions on inlet - fully developed flow
FACE_EQUATION <inlet fully developed> "facegrad(dot(<u[l
   =:]>,cellave[lastface](<facenorm[l=:]>)))" ON <inlet>
    # normal velocity component is fully developed
FACE_EQUATION <inlet noslip> "dot(<u[l=:]>,<facetang1[l
   =:]>)" ON <inlet> # no component tangential to inlet

# set velocity distribution corresponding to fully
   developed Cartesian flow
FACE_EQUATION <inlet flowrate> "<u_f>+6.d0*<u_av>*<cellx
   [l=2]>*(0.41d0-<cellx[l=2]>)/(0.41d0^2)" ON <inlet> #
    specified velocity distribution

# alternatively, set uniform inlet pressure giving
   required average velocity
#NONE_DERIVED <u_av_calc> "facesum(-<u_f>*<facearea>,<
   inlet>)/facesum(<facearea>,<inlet>)" # calculate
   average velocity directed into the domain
#NONE_UNKNOWN <p_in> [Pa] 1.d0 "1.d0" # define the
   pressure at inlet
#NONE_EQUATION <p_in for flowrate> "<u_av_calc>-<u_av>"
   # set flowrate through inlet to give required average
    velocity
```

```
#FACE_EQUATION <inlet flowrate> "<p>-<p_in>" ON <inlet>
   # apply specified pressure over inlet

# calculate drag and lift on object
NONE_OUTPUT <F_drag> [N] "facesum(<facearea>*dot(<
   facenorm[1=:]>,<tau[1=:,1]>),<cylinder>)" # force on
   object in axial direction
NONE_OUTPUT <F_lift> [N] "facesum(<facearea>*dot(<
   facenorm[1=:]>,<tau[1=:,2]>),<cylinder>)" # force on
   object in vertical direction
NONE_OUTPUT <C_drag> "2.d0*<F_drag>/(<rho>*<u_av>^2*0.1
   d0)" # drag coefficient
NONE_OUTPUT <C_lift> "2.d0*<F_lift>/(<rho>*<u_av>^2*0.1
   d0)" # lift coefficient

#
   ------------------------------------------------------------
```