# arb

code version 0.23

Dalton Harvie

October 20, 2010

# 1 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-Ralphson method, so implicitly discretised equations can be solved efficiently; and

- The unstructured mesh over which the equations are solved can be composed 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;

- maxima;

- perl;

- A sparse matrix linear solver: pardiso and various Harwell Subroutine Library routines are supported;

- Some general linear algebra routines: Lapack and the Numerical Recipes in Fortran Library are supported;

- gmsh; and

- The core unix routines `tar`, `gzip` and `rsync`.

Further details about how to get and install this software are given in section 2.2.

arb is copyright Dalton Harvie (2009,2010), but released under the GNU General Public License (GPL). Further details of this license can be found in the license directory once the code has been unpacked.

# 2   Working with arb

## 2.1   Managing arb's files

All routine setup, compiling and running of arb is conducted 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 'transporting' arb and creating this required file/subdirectory structure.

### 2.1.1   Unpacking the code

To create a new version of arb, a new working directory should have been obtained containing the four files

- `archive.tar.gz`

- `unpack`

- `README`

- `LICENSE`

Using the command

```
./unpack
```

from within this directory will unpack the archive, creating the necesary file/sub-directory structure.

### 2.1.2 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 it. Within this directory is a `contributed` subdirectory which should/may contain contributed third party code along with associated arb interface modules, as well as other error handling modules that handle runtime cases where the third party routines are not available.

- `src_equations` directory: contains the perl script `setup_equations.pl` and a template file `equations_module.f90`. These files are used to create the fortran file `equations_module.f90` within `src` which is specific to each problem.

- `obj` directory: the fortran source code (arb) is built within this directory.

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

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

- `doc` directory: contains documentation including this manual.

- `gmsh` directory: generally the gmsh structure (`.geo`) and mesh (`.msh`) files are stored in this directory. Examples that are detailed in this manual (section 5) are stored in the `gmsh/manual` subdirectory.

- `examples` directory: example problem-specific files `equations.in` and `constants.in` are stored here. Examples that are detailed in this manual (section 5) are stored in the `examples/manual` subdirectory.

- `packer` directory: contains various files specific to the pack and unpack operations.

- `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` 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 `src`. `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.

- `setup_equations` link: this is a link to the perl exectuable `setup_equations.pl` within `src_equations`. This allows the equations to be setup from the working directory.

- `arb` script: this shell script is a wrapper script for making and running the `arb` exectuable. Amoung other things, it performs simulation restart (not implemented) and scratch file management.

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

### 2.1.3 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`, `--gmsh` or `--all` specify that all files within either the `example`, `gmsh` or both directories are contained within the archive, rather than just those in the `manual` subdirectories. Including the gmsh option

may be necessary to transport a particular `.msh` or `.geo` mesh file with the other files. Also, by default only source code within the `src/contributed` directory that is not subject to a third party license is included in the archive. Using the options `--contributed` or `--all` causes all files in these directories to be archived.

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

## 2.2 Checking for prerequisit software

A number of third-party applications must be installed in order to use `arb`.

### 2.2.1 Maxima

Equation generation is performed using the Maxima Computer Algebra system, so if you are going to alter the file `equations.in` and run `setup_equations` this software will need to be installed. 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, from which the command `quit();` quits. Otherwise on ubuntu linux maxima can be installed using `apt-get install maxima`. On OsX maxima can be installed using the macports software, using `port install maxima`. Alternatively, a precompiled version for the mac is available from sourceforge (this is the much easier option as of 9/8/10) - if using this package you will need to get the executable command (ie, `Maxima.app/Contents/Resources/bin/maxima`) into your path somehow.

### 2.2.2 Perl

Most likely you already have a version of perl installed.

### 2.2.3 A fortran compiler

Two different compilers have been tested with arb. The Intel Fortran compiler `ifort` is the preferable of the two as it includes the Intel Maths Kernel library

which itself includes the Pardiso routines (see next section), and is generally the faster of the two: however this compiler is not free except on linux and even then, under specific circumstances.

The gnu compiler `gfortran` is free and is available on both the OsX and linux platforms. However, it does not include the Pardiso routines. On ubuntu linux it can be installed with `apt-get install gfortran`. On OsX it can be installed using `port install gfortran`. Binary versions of `gfortran` may also be available.

Compiler choice needs to be specified in `src/makefile` prior to compiling arb. If using `gfortran` set the variable

```
compiler = gnu
```

near the top of the file. If using the intel compiler set

```
compiler = intel
```

instead. `ifort` versions of 11.1.069 and newer have been tested on both linux and OsX. `gfortran` version 4.2 on ubuntu 8.04 doesn't seem to work (internal compiler error) whereas version 4.6 on osX does.

### 2.2.4 Pardiso

Pardiso is a sparse linear matrix solver and is at present the best method available within arb to invert the Newton-Ralphson Jacobian matrix. This 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 output of make command for confirmation).

### 2.2.5 Harwell Subroutine Library

TODO.

### 2.2.6  Numerical Recipes in Fortran 77

TODO.

### 2.2.7  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.

## 2.3  Running a simulation

There are 5 basic steps to running an arb simulation. To demonstrate we show how to simulate the steady-state flow of a Newtonian fluid around a cylinder that is contained in a 2d channel:

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 5.1 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 3.3 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 5.1 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 3 and 4.

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).

# 3 Inside arb

## 3.1 Code structure

## 3.2 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.

## 3.3 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.

### 3.3.1 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.

### 3.3.2 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.

### 3.3.3   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 4) 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 4. Not really a user region. |
| `<downwindfaceicells>` | face | the cell that is downwind of the face, used when performing `faceave[advection]` averaging (see section 4. Not really a user region. |
| `<cellkernel[l=0]>` | cell | surrounding faces used in a cell averaging kernel (see section 4. Not really a user region. |
| `<cellkernel[l=1-3]>` | cell | surrounding cells used in cell derivative kernels (see section 4. Not really a user region. |
| `<cellkernel[l=4]>` | cell | surrounding nodes used in a cell averaging kernels (see section 4. Not really a user region. |
| `<facekernel[l=0-6]>` | face | surrounding cells used in face averaging and derivative kernels (see section 4. Not really a user region. |
| `<noloop>` | face/cell | dummy region which specifies no elements. |

## 3.4   Variable types

There are six types of user defined variables: constant, transient, dependent, independent, 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.

### 3.4.1   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 3.4.9).

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

14

- `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 4.

- `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
```

### 3.4.2 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 3.4.1 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
```

### 3.4.3 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

16

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 3.4.1 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>
```

### 3.4.4 Transient type variable defined in `equations.in`

*Synopsis:*

These will be implemented for version 0.3 (transient).

### 3.4.5 Dependent type variable defined in `equations.in`

*Synopsis:*

Dependent variables depend on the independent variables and other previously defined (ie, above in the file) dependent variables.

*Defining statements:*

```
CELL_DEPENDENT <name> [multiplier*units] "expression" ON
    <region> options # comments
FACE_DEPENDENT <name> [multiplier*units] "expression" ON
    <region> options # comments
NONE_DEPENDENT <name> [multiplier*units] "expression"
   options # comments
DEPENDENT <name> [multiplier*units] "expression" options
    # comments
```

*Statement components:*

Along with the information presented in section 3.4.1, the following applies to dependent variables:

- (CELL_|FACE_|NONE_|)DEPENDENT *(required)*: If no centring is specified then none centring is assumed.

- "expression" *(required)*: This is an expression for the dependent variable in terms of constant, transient, independent, previously defined dependent (appearing above in equations.in) and system variables.

- ON <region> *(optional)*: If ommitted then by default a cell centred dependent will be defined on <all cells> and a face centred dependent on <all faces>.

- options *(optional)*: This is a comma separated list of options. Valid options for dependent variables (as well as those given in section 3.4.1) include:

  - noderivative: Normally the derivative of this variable's expression is calculated with respect to each independent variable (the Jacobian) when performing the Newton-Ralphson 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-Ralphson procedure. Using this option will usually slow convergence.

- **positive/negative**: Including one of these options causes the code to check the sign of the dependent 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_DEPENDENT <tau[l=1,1]> "<p> - <mu>*2.d0*facegrad[l
   =1](<u[l=1]>)" compoundoutput
CELL_DEPENDENT <graddivp[l=1]> "celldivgrad[l=1](<p>)" #
    divergence based pressure gradient
```

### 3.4.6  Independent type variable defined in `equations.in`

*Synopsis:*

Independent variables are those upon which the equations and dependent variables ultimately depend.

*Defining statements:*

```
CELL_INDEPENDENT <name> [multiplier*units] magnitude "
   expression" ON <region> options # comments
FACE_INDEPENDENT <name> [multiplier*units] magnitude "
   expression" ON <region> options # comments
NONE_INDEPENDENT <name> [multiplier*units] magnitude "
   expression" options # comments
INDEPENDENT <name> [multiplier*units] magnitude "
   expression" ON <region> options # comments
```

*Statement components:*

Along with the information presented in section 3.4.5, the following applies to independent variables:

- (CELL_|FACE_|NONE_|)INDEPENDENT *(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 independent variables. This magnitude is used when checking on the convergence of the solution.

- "expression" *(required)*: For an independent variable the expression specifies the variable's initial value. The expression may contain constants, previously defined (initial) dependent values (those appearing above in equations.in) and system variables.

- ON <region> *(optional)*: If ommitted then by default a cell centred independent will be defined on <all cells> and a face centred independent on <all faces>.

- options *(optional)*: The noderivative option is not applicable for independent variables.

*Examples:*

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

### 3.4.7 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 independent variables. Furthermore, for the system to be well posed the equations must be independent (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 3.4.5, 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, dependent, independent and system variables.

- ON <region> *(optional)*: If ommitted then by default a cell centred equa-
  tion 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
```

### 3.4.8 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 3.4.5, 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, dependent, independent, 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[l=:]>,<tau[l=:,1]>),<cylinder>)" # force on
   object in axial direction
```

### 3.4.9  System variables

## 3.5  Simulation options

## 3.6  Data visualisation

# 4  Expression language reference

## 4.1  Operators

### 4.1.1  `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 $V_{\text{cell}}$ and $S_{\text{cell}}$ 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[l=:]>,<facenorm[l=:]>))
```

where in this case the vector $\boldsymbol{u}$ is represented by the three component variables `<u[l=1]>`, `<u[l=2]>` and `<u[l=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 independent variables for the Newton-Ralphson 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
```

**4.1.2**  `cellgrad` **or** `facegrad`: **Gradient evaluated at a cell or face**

**4.1.3**  `celldivgrad`: **Gradient evaluated at a cell calculated via a divergence**

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

**4.1.5**  `celllimiter`: **Gradient limiter for ensuring advection stability**

**4.1.6**  `cellif` **or** `faceif` **or** `noneif`: **If conditional statement**

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

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

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

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

# 5  Example applications

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

Listing 1: ../../gmsh/manual/channel_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};
```
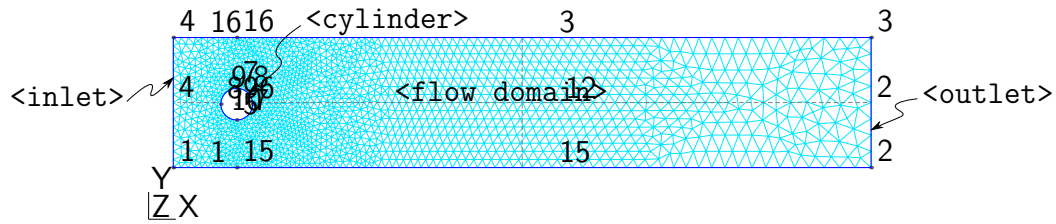
Figure 1: Geometry and resulting mesh from the `surface.geo` file for the flow through a 2d channel containing a cylinder problem.

```
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 0.23
```

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

# 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 "gmsh/manual/channel_with_cylinder/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

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

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 0.23
#
   -----------------------------------------------------------------
```

```
# 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|) DEPENDENT <name> [units] magnitude
   "expression" ON <region> options # comments
# (CELL_|FACE_|NONE|) INDEPENDENT <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 DEPENDENT, EQUATION
#   positive,negative - for DEPENDENT, INDEPENDENT,
   EQUATION
#   harmonic - for CONSTANT, TRANSIENT, DEPENDENT,
   INDEPENDENT
#   compoundoutput/nocompoundoutput - for ALL
#   componentoutput/nocomponentoutput - for ALL

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

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

```
FACE_DEPENDENT <tau[l=2,1]> "<tau[l=1,2]>"

# a Rhie-Chow-type correction is applied to the face
   velocities
CELL_DEPENDENT <graddivp[l=1]> "celldivgrad[l=1](<p>)" #
    pressure gradient calculated via a divergence (
   consistent with momentum conservation)
CELL_DEPENDENT <graddivp[l=2]> "celldivgrad[l=2](<p>)" #
    pressure gradient calculated via a divergence (
   consistent with momentum conservation)
FACE_DEPENDENT <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_DEPENDENT <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_DEPENDENT <u_f> "dot(<u[l=:]>,<facenorm[l=:]>) + <
   u_f_{correction}>" # volume (velocity) transport
FACE_DEPENDENT <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_DEPENDENT <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_DEPENDENT <u_av_calc> "facesum(-<u_f>*<facearea>,<
   inlet>)/facesum(<facearea>,<inlet>)" # calculate
   average velocity directed into the domain
#NONE_INDEPENDENT <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[l=:]>,<tau[l=:,1]>),<cylinder>)" # force on
   object in axial direction
NONE_OUTPUT <F_lift> [N] "facesum(<facearea>*dot(<
   facenorm[l=:]>,<tau[l=:,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

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