arb

code version 0.54

Dalton Harvie

August 12, 2015

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-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 2.3.

2 Installation

2.1 No-nonsense setup on ubuntu 10.04 or 12.04:

1. Install required packages via `apt-get`:

   ```
   sudo apt-get install maxima maxima-share gfortran
     liblapack-dev libblas-dev gmsh curl gnuplot paraview valgrind
   ```

2. Download the arb files, unpack the archive, move to the working directory and create the directory structure (see section 2.4 for more detail)

   ```
   wget http://people.eng.unimelb.edu.au/daltonh/
     downloads/arb/code/latest.tar
   tar -xf latest.tar
   cd arb_*
   ./unpack
   ```

3. Download and compile the suitesparse solver (see section 2.3.4 for more detail)

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

4. Run a test simulation and view the results (see section 3 for more detail)

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

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

2.2 No-nonsense setup on OsX:

Installation on OsX is slightly more fiddly. This procedure is known to work on OsX 10.6 and 10.8, and previously I had it working on OsX 10.4.

1. Install Apple's Xcode developer package.
2. Install a gfortran binary package as per section 2.3.2
3. Install a maxima binary package as per section 2.3.1
4. Follow steps 2 and onwards from the ubuntu instructions.

2.3 Installing prerequisit software (in more detail)

2.3.1 Maxima

Equation generation is performed using the Maxima Computer Algebra system. It is released under the GNU General Public Licence (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.

2.3.2 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 as currently also supports OMP execution. 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 (except for the OMP caveat).

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 (including Composer XE) have been tested on both linux (v12 and v13) and OsX (v12 only).

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. The version on ubuntu 12.04 is fine too. 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.

2.3.3 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). There is currently no interface to use this solver external to the Intel Math Kernel library.

2.3.4 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 version 3.6.0 of UMFPACK and version 4.0.3 of METIS (using curl - install on ubuntu using `sudo apt-get install curl` if you don't have it already), 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 2.3.2). 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 from within the `suitesparse` directory.

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

- UMFPACK.tar.gz 02-Jun-2010 11:46 and metis-4.0.tar.gz (4.0.1)
- SuiteSparse-3.6.0.tar.gz and metis-4.0.tar.gz (4.0.3)

2.3.5 Perl

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

2.3.6 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!).

2.3.7 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.

2.3.8 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 Licence (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 10.04/12.04:

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

On earlier ubuntu versions the repository version is too old.

Installing `gmsh` on anything else or getting the latest version on ubuntu:

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.

2.3.9 ParaView

There is now capability to output in the `.vtk` format, used by (for example) ParaView. To install ParaView on ubuntu use:

```
sudo apt-get install paraview
```

On the mac there are binaries available. Input of this file format is not supported.

2.3.10 Tecplot

Tecplot is proprietary visualisation software. `arb` can output to its `*.dat` asci format, but not input from this format.

2.4 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.

2.4.1 Unpacking the code

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

```
archive.tar.gz
unpack
readme
licence
version
```

Using the command

```
./unpack
```

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

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

2.4.2 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_template.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. Some notable files are `equations_module.f90` which contains all of the fortran coding that is specific to the current problem, and `fortran_input.arb` which is the only input file read by the executable fortran file.
- `tmp` directory: temporary files are stored within this directory.

- `tmp/setup` directory: includes files produced during the setup of the problem (running of `setup_equations.pl`). If you're having to debug a problem setup then this is the place to look. The file `debugging_info.txt` contains a lot of detail regarding the equation setup. The file `unwrapped_input.arb` is a completely unwrapped version of the last run input, in which any `INCLUDE`d files are unwrapped, with any relevant text strings substituted. This file can be input directly to arb again if desired.

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

- `misc` directory: contains miscellaneous files — for example: a `create_mesh` script which builds a `.msh` file from any `.geo` file in the working directory; a `batcher.pl` script which automates the running of consecutive arb runs; and a `test_suite` script which runs all cases in the `examples` directory as a check.

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

- `examples` directory: example problem-specific `*.arb` files with their associated geometry files (structure files with `.geo` extension, and mesh files with `.msh` extension) are stored here. Looking through these examples is currently the best way to learn about the language syntax and generally understand how to run arb simulations.

- `templates` directory: New for version 0.4, this directory contains template `*.arb` files for doing common chunks of setup. For example, there is a `navier_stokes` directory which contains all of the code chunks necessary to run Navier-Stokes problems. There are also directories to implement high-order limited advection in both two and three dimensions. These pieces of template code can be used via the `INCLUDE_ROOT` and `INCLUDE` commands.

- `pack` script: packs the directory for transportation to another location or computer. This script has many useful options — try `./pack --help`.

- `licence` directory: contains licence details, including the (GNU GPL) licence under which arb is released and the specific version of the code. A history file within the directory `packer_history.txt` records the distribution history of this particular code directory.

- `*.arb` files: these files and an associated `.msh` or `.geo` file contain all the problem-specific information required for a particular simulation.

- `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, whether using OMP or not, whether you are restarting a simulation or starting from scratch. This script works out when the equation meta-programming has to be redone or not, and when recompilation is necessary, although with other options you can overwrite this behaviour. To list the options type `arb --help`.

2.4.3 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_v[version]_[date]` 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 licence 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 and example input files will be copied to the working directory. 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).

Use `./pack --help` to list other options.

3 Running simulations

3.1 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

```
heat_conduction_around_ellipse.arb
surface.geo
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_around_ellipse` to the working directory.

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`. Note that if a specific `.arb` file is not passed to the `arb` script, as above, it will default to any `.arb` files that exist within the working directory. The above code worked because only one `.arb` file should have been present in the working directory when the code is first unpacked.

To view the output type

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

You should see the temperature field around a heated ellipse.

3.2 Other examples:

Other example simulation files are included in the subdirectories of `examples`. To run any of these first copy the relevant `*.arb` and `*.geo` or `*.msh` files to the working directory. If only a `*.geo` file is included (or if you want to change the mesh resolution) create the `.msh` file via the included `create_mesh` script:

```
misc/create_msh/create_msh geo_file.geo
```

replacing '`geo_file.geo`' with name of the file. Note that an optional first numeric argument can be passed to the script to control the relative cell size — the smaller the parameter, the smaller the mesh element size and greater the number (see `misc/create_msh/create_msh --help`). Finally run arb using `./arb` and view the results.

Right now the examples include:

- `1d_nonuniform_diffusion`: diffusion along a line containing two diffusion coefficients (1D)
- `advection_2d_box_test`: advection of scalar in two dimensions demonstrating high-order limited advection inclusions
- `bouncing_ball`: a transient bouncing ball
- `cube_laplacian_dhctac10_2012`: nonlinear laplacian in a 3D box, presented at the CTAC10 conference (and shown in associated paper [**?**])
- `heat_conduction_around_ellipse`: diffusion (nominally heat) around an ellipse within a box (2D)
- `heat_conduction_with_linked_regions`: similar to the above but demonstrating the use of linked regions (2D)
- `inviscid_burgers_equation`: 1D transient
- `laplacian_in_square`: comparison between the numerical and analytical solution for a laplacian problem (2D)
- `slit_flow_with_structured_mesh_xy_plane`: Navier-Stokes flow with structured mesh
- `slit_flow_with_structured_mesh_xz_plane`: Same but in different direction
- `steady_state_channel_flow_with_cylinder`: CFD benchmark problem of fluid flow around a slightly offset cylinder at Re $= 20$ (2D)
- `stokes_flow_through_cylinder`: Stokes flow through a pipe in 3D
- `weiluns_stokes_sphere`: Flow around a sphere in cylindrical polar coordinates

3.3 The `arb` run script:

This shell script handles the perl meta-programming, fortran compiling, linking and running of arb. It uses a makefile (`src/makefile`) to determine when various components have to be remade. All code building is done in the directory `build`.

By default the `arb` script will try to do all that is necessary to run a simulation from scratch (except make the suitesparse libraries), however its behaviour can be altered by (at least) the following options:

- `--clean`: remove all meta-programmed fortran source and executable objects before building again.
- `--clean-compile`: remove all executable objects before building again. This may be handy if you want to run a simulation on a machine that has a fortran compiler, but not say maxima.

- `--clean-setup`: remove all meta-programmed fortran source before building again.
- `--setup`: create perl meta-programmed fortran source (default).
- `--no-setup`: do not recreate perl meta-programmed fortran source.
- `--compile`: create executable objects by compiling fortran source (default).
- `--no-compile`: do not recreate executable objects by compiling fortran source. If both `--no-setup` and `--no-compile` are specified, the existing `arb` executable will be rerun regardless of changes to the `.in` files.
- `--run`: run the `arb` executable once it is built (default).
- `--no-run`: do not run the `arb` executable.
- `-q|--quiet`: send screen output to `output/output.scr`.
- `-c|--continue`: continue on from a previous simulation. This is the not the default behaviour. Unless this is specified the contents of `output` will be cleared out before each simulation. Note that a copy of the previous output files are placed in `output/previous` so if you accidentally run `arb` without the continue flag (once) you can get the old files back from this directory.
- `-d|--debug`: compile the fortran using debug flags, and run the executable within the `gdb` environment.
- `--compiler-gnu`: compile the fortran using `gfortran` (default if `ifort` is not present).
- `--compiler-intel`: compile the fortran using `ifort` (default if `ifort` is present).

3.4 A more detailed guide: Newtonian fluid flow

To discuss the working method in more detail we use the example of the steady-state flow of a Newtonian fluid around a cylinder that is provided in `examples/steady_state_channel_flow_with_c`

There are 5 basic steps to setting up, running and debugging 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. 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. You can either do this via the gmsh GUI, or via the simple `create_mesh` script (mentioned above) which calls gmsh. When creating a 3D mesh be sure to optimise the mesh after creation (greatly improves mesh quality - `create_mesh` does this by default for 3D meshes).

arb uses the concept of 'regions' to locate various equations (for example boundary conditions, domain equations etc) and these should be defined in the `.geo` file prior to meshing (some regions are also created by arb). 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). Like variable names (discussed later) region names can contain any characters except for `"` and `#`. Region and variable names are case sensitive.

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 4.2 for more details.

2. **Edit the** `*.arb` **input file**: Aside from the mesh file, all information that is specific to a simulation is contained in this file. Some commands/statements/options within this file will require the fortran to be recreated and the fortran executable to be recompiled, while other commands/statements/options within this file are passed directly to the fortran exectuable at run time (see `build/fortran_input.arb`).

   Further details regarding the syntax of the `*.arb` file can be found in Sections 4 and 5. As these are not very complete, it is a good idea to look through the example files too.

3. **Run** `arb`**:** From the working directory

   ```
   ./arb
   ```

   will run arb.

   The first thing that happens is that the makefile checks what libraries/software are available and prints diagnostic messages to the screen.

   The perl script `setup_equations.pl` is then called. It reads any `*.arb` files that are present in the working directory, and using maxima, creates the fortran source code file `build/equations_module.f90`. The name of a specific `*.arb` file can also be given, as in

   ```
   ./arb steady_state_channel_flow_with_cylinder.arb
   ```

   to only run this one simulation. `setup_equations.pl` writes plenty of progress information to the screen - if all goes well this output will end with the statement '`SUCCESS`'. Otherwise errors in the `*.arb` file will need to be found and corrected. Aside from the screen output, more debugging information is written by `setup_equations.pl` to the file `tmp/setup/debugging_info.txt`. Other files in the `tmp/setup` directory trace the interaction between the perl script and maxima.

   Next the fortran code is compiled. If the debugging option has been requested (`./arb --debug`) the any error messages will be contained in `build/make.log`.

If all goes well the resulting executable will run. Alongside the `.msh` and `output.stat` (statistics) files written to the directory `output`, other files in this directory can be used to diagnose problems. If these don't help, there are many `debug` and `debug_sparse` logicals scattered through the code, and in `src/general_module.f90` there are options to print out alternative output files.

4. **View results:** arb produces a file `output/output.msh` which can be opened by gmsh for viewing. The file `output/output_step.csv` is also produced which contains (by default) mainly `none` centred variables, and can be easily graphed using the script `misc/plot_step/plot_step.pl`.

5. **Rerun the simulation:** The `output/output.msh` file produced by `arb` includes both the mesh information as well as variable data, so can be passed back to arb as an input `.msh` file for subsequent simulations. To do this replace the existing `MSH_FILE` line in the `*.arb` file with

   ```
   MSH_FILE "output/output.msh" input
   ```

   and rerun arb, without first removing this file via

   ```
   ./arb --continue
   ```

   If you forget the `--continue` flag then you can recover the previous output files that are saved in `output/previous` for just this emergency.

4 Simulation setup reference

4.1 Meshes

arb uses an unstructured mesh composed of cell elements that are separated by face elements. The dimension of each element is specified on a per-element basis, consistent with the particular computational domain that the element is within (that is, not globally). Cell elements are classified as either boundary cells (that is, on the boundary of a domain) or domain cells (that is, contained within a domain). Domain cells have a dimension that is equal to that of the domain they are in (ie, dimension 3/2/1 if the domain is a volume/surface/line, resp.). Boundary cells have a dimension that is one less than that of the associated domain (ie, dimension 2/1/0 if the domain is a volume/surface/line, resp.). Face elements are any elements that separate cell elements. Face elements also have a dimension that is one less than that of the domain they are within. Some face elements are specified explicitly within a `.msh` file (if they are part of a physical entity such as `<inlet>` for example), while the remainder are generated by arb when the mesh is read in. Face elements are also classified as being either domain faces or boundary faces. Each boundary face has the same geometry, and is conincident with, a boundary cell. Hence, a mesh has the same number of boundary faces as boundary cells.

Meshes are read in from `.msh` files, generally produced by the gmsh program. Multiple `.msh` files can be read in by arb for each simulation. arb has been coded to be able to handle any poly-sided first order elements supported by the gmsh file format. It has been tested to date (v0.3) with tetrahedron, boxes and prisms in 3D, triangles and rectangles in 2D, lines in 1D and points in 0D. Tetrahedron, triangles, lines and points are the default element geometries created by gmsh.

Meshes and data are also exported by arb using the `.msh` format. During every simulation all domains and all output-enabled data will be written by default to the `output/output.msh` file. Other files may also be written, corresponding to any `.msh` files that are read in (with any associated output-enabled data). Regions imported from `.msh` files as well as regions created by arb will be exported to any written `.msh` files, however note that as the physical entities handled by gmsh can only have a single dimension, elements that have a dimension that is less than any others within a region will not be associated with that region in any arb-created `.msh` files. This is relevant for example when a compound region is created that contains both domain and boundary cells. When this arb-written `.msh` file is displayed by gmsh it will only appear to contain the domain cells.

4.1.1 Cell and face element specification

The distinction between cell elements and face elements is not made by gmsh or contained explicitly in the `.msh` file, but rather must be made by arb when a `.msh` file is read in. Gmsh's behaviour is to only write an element to a `.msh` file if it is a member of a physical entity. Further, each physical entity has a single dimension. So, to decide whether an element is either a face or cell element, arb does two things when reading in each `.msh` file:

1. The maximum dimension of all physical entities with the `.msh` file is found. This is stored as the dimension of the particular mesh;
2. When an element is read in that has the same dimension as that of the mesh, it is regarded as a cell element. If it has a dimension that is one less than that of the mesh, it is regarded as a face element. If it has a dimension that is two or more less than that of the mesh, then the element is ignored.

So what's the implication of all this? Generally arb will be able to work out from each `.msh` file which elements within it are cell elements and which are face elements. The only time it won't is when *there are multiple domains having different dimensions contained within the one .msh file*. For example, you have both a volume domain and a surface domain specified within a `.msh` file, on which you want separate (but possibly linked) sets of equations solved.

If you do have multiple domains having different dimensions contained within the one file then the dimension of all regions (that is, physical entities) contained within the `.msh` file that belong to any domains that have a dimension that is *less than* that of the `.msh` file need their centring explicitly specified. For example, if a `.msh` file contains both a volume and a surface domain, then all regions associated with the surface domain must have their centring explicitly specified. Statements for specifying this cell/face centring for particular regions (gmsh physical entities) are described in section 4.2.1. Alternatively, there is another way that may work for your simulation: As arb can read multiple `.msh` files for each simulation, it may be easier to place domains of different dimensions in separate `.msh` files. The cell/face specification will then be handled automatically without additional statements in the `constants.in` file (If you want multiple domains to share common mesh features however this may be difficult to accomplish using gmsh).

4.1.2 Data and mesh file rereading

The `.msh` files written by `arb` can contain data. Variables associated with cell elements can be written in either `ElementData` (a uniform value for each cell) or `ElementNodeData` (values vary linearly within each cell) gmsh formats. Variables associated with face elements will only be written in `ElementData` format. Variables which are `none` centred are written using a special `Data` format which gmsh won't display. Note that face and cell boundary elements are not written separately to each `.msh` file by arb, but rather as a single element. Hence both cell and face boundary data is associated with a single element in each `.msh` file.

One purpose of exporting data to a `.msh` file is to provide initial conditions for another (or next) simulation. In this case generally you just have to specify the `output/output.msh` file from the previous simulation as the mesh file to be read in for the next simulation. Note that each arb-written `.msh` file contains all the information about a mesh that was originally contained in the mesh-only gmsh-written .msh file: Hence, when starting a simulation from an arb-written datafile is it not necessary (nor does it make sense) to also read in the original gmsh .msh file. Also note that data files that contain variable values can only refer to mesh elements that are specified in the same data file (unless some fancy facelink magic is worked in your equations).

4.1.3 Mesh read and write options

Mesh and data input and output is specified by MSH_FILE statements within the constants.in file:

```
MSH_FILE "msh_file_name_including_path" comma,separated,list,
   of,options # comments
```

The file name refers to the read location. Options for the default output file should be referred to by output/output.msh (which is the read location if it did exist). If a .msh file is to be written it will always be written to the output directory. As a result, all file basenames must be unique.

Three types of options are available for each .msh file:

Output options:

These options specify what information is to be written to the .msh file.

- output: Both a mesh and all specified variables will be written.
- centringoutput: Both a mesh and all specified variables will be written. Output will be split between three files, each containing variables of only a single centring (cell, face and none). This can be handy for gmsh compatibility when doing cutgrid and streamtrace operations for example.
- centringmeshoutput: Only the mesh is written, split between three files as above.
- meshoutput: Only the mesh is written.
- nooutput: Neither the mesh or any data will be written.
- vtkoutput,centringvtkoutput,meshvtkoutput,centringmeshvtkoutput,novtkoutput: Same as the *.msh file options, but for *.vtk output, compatible with ParaView (for example). The default is novtkoutput.
- datoutput,centringdatoutput,meshdatoutput,centringmeshdatoutput,nodatoutput: Same as the *.msh file options, but for *.dat output, compatible with Tecplot. The default is nodatoutput.

By default all meshes have the `nooutput` option specified, with the exception of the `output/output.msh` mesh, which has option `output`.

Input options:

These options specify what information is to be read from the `.msh` file.

- `input`: Both a mesh and all relevant data will be read.
- `centringinput`: Both a mesh and all relevant data will be read. In this case the existing `.msh` is split into three, each containing variables of only a single centring (cell, face and none) as output from a previous simulation employing `centringoutput`. In this case the filename should be specified without the centring, for example `output/output.msh` (rather than `output/output.cell.msh`).
- `centringmeshinput`: Only the mesh is read, split between three files as above.
- `meshinput`: Only the mesh is read.
- `noinput`: Neither the mesh or any data will be read.

By default all meshes have the `input` option specified, with the exception of the `output/output.msh` mesh, which has option `noinput`.

Data format options:

These options specify how cell centred data will be written to the `.msh` file. These options overwrite any data format options specific to individual variables.

- `elementdata`: All cell data will be written using the `ElementData` format.
- `elementnodedata`: All cell data will be written using the `ElementNodeData` format.
- `elementnodelimiteddata`: All cell data will be written using the `ElementNodeData` format, but with the gradients in each cell limited so that each vertex value is bounded by surrounding cell values.

By default all meshes have no data format options specified, the format instead being determined by the options contained in the individual variable definitions within input `.arb` file.

4.2 Regions

Regions are sets of 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 may contain elements of different dimensions (see caveat in previous section regarding gmsh display of this though). Regions can be defined by the user directly in gmsh when the mesh is generated, or via statements in the `.arb` file that are interpreted when `arb` is run. There are also several generic system generated regions. Region names must be delimited by the <> characters, and within these delimiters cannot contain the characters <, >, # or ". Apart from these four characters their names may contain any non-alphanumeric characters.

4.2.1 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 that the physical entity is visible.

You can specify the cell or face designation of any gmsh element using the following commands within the `constants.in` file. This is seldom necessary (although it doesn't hurt either), unless the `.msh` file it is contained within contains multiple domains, of differing dimensions (see discussion in previous section).

```
CELL_REGION <gmsh_region_name>
FACE_REGION <gmsh_region_name>
```

4.2.2 Defining regions within the `constants.in` file

There are several types of region specification statements that can be used in the `constants.in` file. Regions specified by these statements will overwrite any regions defined in the `.msh` files, however a warning is issued (This allows `.msh` files to be reread without altering region definition statements). The specification statements are:

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
CELL_REGION <name> "AT x1 x2 x3 PART OF <domain>" # comments
FACE_REGION <name> "AT x1 x2 x3" # comments
FACE_REGION <name> "AT x1 x2 x3 PART OF <inlet>" # comments
```

Search ref: AT region in `setup_module.f90`

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. An optional `PART OF <a region>` confines the choice of an element to those within `<a region>`. In this case `<a region>` must have the same centring as the region statement.

Within box region:

```
CELL_REGION <name> "WITHIN BOX x1_min x2_min x3_min x1_max
   x2_max x3_max" # comments
FACE_REGION <name> "WITHIN BOX x1_min x2_min x3_min x1_max
   x2_max x3_max" # comments
```

Search ref: WITHIN region in `setup_module.f90`

This statement defines a region including all elements (cell or face) that lie within a box with faces orientated with the coordinate directions, and location defined by the two corner points having the minimum (`x1_min x2_min x3_min`) and maximum (`x1_max x2_max x3_max`) coordinate values. An optional `PART OF` function is planned.

Boundary of region:

```
CELL_REGION <name> "BOUNDARY OF <region>" # comments
FACE_REGION <name> "BOUNDARY OF <region>" # comments
```

This statement defines a region that contains *only* the boundary elements (either cell or face) that border the region `<region>`.

Domain of region:

```
CELL_REGION <name> "DOMAIN OF <region>" # comments
FACE_REGION <name> "DOMAIN OF <region>" # comments
```

This statement defines a region that contains *only* the domain elements (either cell or face) that are associated with the region `<region>`.

Associated with region:

```
CELL_REGION <name> "ASSOCIATED WITH <region>" # comments
FACE_REGION <name> "ASSOCIATED WITH <region>" # comments
```

This statement defines a region that contains *both* the domain and boundary elements (either cell or face) that are associated with the region `<region>`. Effectively this is a combination of the `BOUNDARY OF` and `DOMAIN OF` statements.

4.2.3 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 5) 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)
`<adjacentfaceupcell>`	face	cell that is adjacent to the current face in the direction of the normal
`<adjacentfacedowncell>`	face	cell that is adjacent to the current face in the opposite direction to the normal
`<upwindfaceicells>`	face	the cell that is upwind of the face, used when performing `faceave[advection]` averaging (see section 5. Not really a user region.
`<downwindfaceicells>`	face	the cell that is downwind of the face, used when performing `faceave[advection]` averaging (see section 5. Not really a user region.
`<cellkernelregion[l=0]>`	cell	surrounding faces used in a cell averaging kernel (see section 5. Not really a user region.
`<cellkernelregion[l=1-3]>`	cell	surrounding cells used in cell derivative kernels (see section 5. Not really a user region.
`<cellkernelregion[l=4]>`	cell	surrounding nodes used in a cell averaging kernels (see section 5. Not really a user region.
`<facekernelregion[l=0-6]>`	face	surrounding cells used in face averaging and derivative kernels (see section 5. Not really a user region.
`<noloop>`	face/cell	dummy region which specifies no elements, or the last element used in an operator's context.

4.3 Variables

This section needs some rewritting: there is only one input file now

There are eight types of user defined variables: constant, transient, derived, unknown, equation, output, condition and local. 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.

4.3.1 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 except for double quotation marks " (which demarcate the expression strings) and hash character # (which indicates that a comment follows). 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 4.3.11).

- 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 5.

- `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 omitted then by default a cell centred constant will be defined on `<all cells>` and a face centred constant on `<all faces>`. Note that referring 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. Options earlier in the list take precedence over later ones. Valid options for the constant variable type include:

  - `output`: The compound variable that this component is a member of to be written to each applicable `.msh` file. The opposite option `nooutput` exists. Default is `output` for unknown variables, output variables, derived cell-centred variables, and transient variables that do not correspond to the oldest stored timestep (that is, `rstep` < `rstepmax`). The default is `nooutput` for everything else.
  - `stepoutput`: The compound variable that this component is a member of is to be included in the `output.step` file. The opposite option `nostepoutput` exists. By default only this option is set only for unknown, output and transient non-centred variables that are at the current timestep (that is, `rstep` $= 0$). The option `stepoutputnoupdate` also exists which specifies that the variable should be included in the `output.step` file, but that its value is not updated before being printed. This option is useful for outputing variables which should only be updated when a `.msh` file is actually written (for example, a variable that records the time when output occurs).
  - `componentoutput`: This component to be written to each applicable `.msh` file. Default is `nocomponentoutput` for all variables.
  - `input`: The compound variable that is component is a member will be read from each applicable `.msh` file. The opposite option `noinput` exists. Default is `input` for all unknown and transient variables and `noinput` for everything else.
  - `componentinput`: This component to be read from each applicable `.msh` file. Default is `nocomponentinput` for all variables.
  - `elementdata`: This compound will be written using the gmsh `ElementData` format. Any data format options specified for each `.msh` file will overwrite this option. Other options include `elementnodedata`, `elementnodelimiteddata`, `componentelementdata`, `componentelementnodedata` and `componentelementnodelimite`
  Default is `elementnodedata` and `componentelementdata`.

Search ref: default output
in `setup_module.f90`

Search ref: default stepoutput
in `setup_module.f90`

Search ref: default input
in `setup_module.f90`

- `comments` *(optional)*: Anything written beyond the first # appearing on each line of the input file 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
```

4.3.2 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 4.3.1 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
```

4.3.3 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 4.3.1 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>
```

4.3.4 Transient type variable defined in `equations.in`

Synopsis:

Transient variables are used only in transient simulations, and are evaluated at the start of each timestep. Transient variables are typically used to store previous timestep values, or to provide constant data to a simulation that depends explicitly on the time.

Defining statements:

```
CELL_TRANSIENT <name> [multiplier*units] "initial expression"
    "expression" ON <region> options # comments
FACE_TRANSIENT <name> [multiplier*units] "initial expression"
    "expression" ON <region> options # comments
NONE_TRANSIENT <name> [multiplier*units] "initial expression"
    "expression" options # comments
NONE_TRANSIENT <name> [multiplier*units] "" "expression"
   options # this will use the update expression as the
   initial expression
NONE_TRANSIENT <name> [multiplier*units] "expression" options
    # the initial expression here depends on rstep
```

Statement components:

Along with the information presented in section 4.3.1, the following applies to transient variables:

- `(CELL_|FACE_|NONE_|)TRANSIENT` *(required)*: If no centring is specified then none centring is assumed.
- `<name>` *(required)*: Along with the rules detailed in section 4.3.1, transient variables are associated with particular relative timesteps. Relative timesteps, described using the term `rstep` in this document, indicate how many timesteps previous to the current one the variable refers to. The `rstep` value of a variable is defined in a similar manner to the direction of a variable, using an `r` index in square brackets at the end of the variable name: For example, `<t[r=0]>` would be the time corresponding to the end of the current timestep, `<t[r=1]>` would be time from the previous timestep, `<t[r=2]>` the time from the (earlier) timestep before that one and so on. If an `r` index is omitted from a definition, then `r=0` is assumed. Actually, any type of variable can be associated with any particular relative timestep, but it is rare to do this with anything other than a transient variable.
- `"initial expression"` *(optional)*: This expression is applied once (only) at the start of a simulation, and represents the variable's initial condition. These initial expressions are applied in the order of increasing `rstep` (relative timestep), meaning that the current (latest) time value is calculated first, followed by the previous timestep value, and then one before etc. This expression should not depend on any transient variables that have a higher `rstep` (an earlier timestep) or that are from the same timestep (equal `rstep`) but defined later in the input file. If an initial expression is not given at all (no quotation marks present for this field), then the value of zero is assumed if the variable has `rstep=0`, or the update expression otherwise. If the initial expression is not specified but quotation marks are present for this field, then the update expression is substituted for the initial expression - ie, a shorthand way of repeating the update expression.
- `"expression"` *(required)*: This expression for the transient variable is applied once at the start of each timestep. These expressions are applied in the order of

decreasing rstep (relative timestep), meaning that the earliest time value is calculated first, followed by the next timestep value, until the current time (rstep $= 0$) is reached. Circular references are not allowed in the expression (in practice this is not limiting).

- ON <region> *(optional)*: If ommitted then by default a cell centred transient will be defined on <all cells> and a face centred transient on <all faces>.
- options *(optional)*: This is a comma separated list of options. Valid options for transient variables are the same as those for constants, as detailed in section 4.3.1)

Examples:

```
NONE_TRANSIENT <t[r=0]> "0.d0" "<t[r=1]>+<dt>" # current end-
   of-timestep time (r=0)
NONE_TRANSIENT <t[r=1]> "<t>-<dt>" "<t>" # time at last step
   (r=1)
NONE_TRANSIENT <t[r=2]> "<t[r=1]>-<dt>" "<t[r=1]>" # time at
   step before last step (r=2, assuming a constant dt)
NONE_TRANSIENT <z[r=1]> [m] "<z>-<w_0>*<dt>" "<z_real>" #
   position of ball at last step (r=1)
NONE_TRANSIENT <w[r=1]> [m/s] "<w>" "<w_real>" # velocity of
   ball at last step (r=1)
```

4.3.5 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 4.3.1, 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 4.3.1) 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. In theory this could be used 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. In practice using these types of limiting conditions to prevent equation singularities slows convergence to an unfeasibly slow rate. It is usually better to choose the form of the equations so that they are stable even for small unphysical excursions, and then check once convergence has been achieved that the results are physical.

Examples:

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

4.3.6 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 4.3.5, 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 constant variables, derived variables, previously defined unknown variables, (initial) transient variables 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[l=1]> 1.d0 "<u_av>" # a velocity component
CELL_UNKNOWN <p> [] 1.d0 "1.d0-<cellx[l=1]>" # pressure
NONE_UNKNOWN <p_in> [Pa] 1.d0 "1.d0" # the pressure at the
   inlet
```

4.3.7 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 4.3.5, 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
```

4.3.8 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 4.3.5, 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[l
   =:]>,<tau[l=:,1]>),<cylinder>)" # force on object in axial
    direction
```

4.3.9 Condition type variable defined in `equations.in`

Synopsis:

Condition variables control the running of the simulation. They can initiate the following actions: output, stop, convergence and a bell.

Defining statements:

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

```
NONE_CONDITION <name> [multiplier*units] "expression" options
    # comments
CONDITION <name> [multiplier*units] "expression" options #
   comments
```

Statement components:

Along with the information presented in section 4.3.5, the following applies to condition variables:

- `(CELL_|FACE_|NONE_|)OUTPUT` *(required)*: If no centring is specified then none centring is assumed.
- `"expression"` *(required)*: For a condition variable, if the evaluated expression is positive (> 0) then the condition is satisfied and the corresponding action will take place. Note that an action will take place if any of the condition variables that correspond to it are positive (in fact, after one positive value is found the remainder are not even evaluated).
- `ON <region>` *(optional)*: If ommitted then by default a cell centred condition variable will be defined on `<all cells>` and a face centred condition variable on `<all faces>`.
- `options` *(optional)*: In addition to the options discussed for the other variables, one or more of the following options may be applied to each condition variable to specify what action it corresponds to:
  - `convergencecondition`: For transient and steady-state simulations, indicates when the Newton loop has converged. Is evaluated at the start of each Newton loop.
  - `stopcondition`: For a transient simulation, indicates when the simulation should finish. Is evaluated at the end of each successful timestep.
  - `outputcondition`: For a transient simulation, indicates when the `.msh` output files should be written. Is evaluated at the end of each successful timestep.
  - `bellcondition`: For a transient simulation, indicates when a noise should be made (this one's a bit silly). Is evaluated at the end of each successful timestep.

Examples:

```
NONE_CONDITION <time based stop condition> "<t>-<tend>"
   stopcondition # when this becomes true (>0.) the
   simulation stops
NONE_CONDITION <bouncing bell> "noneif(<z>,-1.d0,1.d0)"
   bellcondition # is positive when <z> is negative at the
   end of a timestep
```

```
NONE_CONDITION <output test> "<t>-<tout>-<dtout>"
   outputcondition # this will be true (>0.) whenever we are
   <dtout> from last output
```

4.3.10 Local type variable defined in `equations.in`

Synopsis:

Local variables are like derived variables, except that they are not stored, but rather evaluated only when required. Local variables may be used instead of derived variables to save memory. This strategy makes sense if the variable is only going to be used once or twice at each location. Local variables may also be used to split up an otherwise long expression into smaller (and possibly common) sub-statements, dependent on the local conditions. For example, in the examples given below, local variables are used to calculate the second derivative of the normal velocity to a wall, in the normal direction to a wall.

Defining statements:

```
CELL_LOCAL <name> [multiplier*units] "expression" ON <region>
    options # comments
FACE_LOCAL <name> [multiplier*units] "expression" ON <region>
    options # comments
NONE_LOCAL <name> [multiplier*units] "expression" options #
   comments
LOCAL <name> [multiplier*units] "expression" options #
   comments
```

Statement components:

Along with the information presented in section 4.3.5, the following applies to condition variables:

- `(CELL_|FACE_|NONE_|)OUTPUT` *(required)*: If no centring is specified then none centring is assumed.
- `"expression"` *(required)*: A local variable may depend on 'local' variables which correspond to the locale of the calling statement: For example, in the following examples we refer to the `<facenorm>` of the face on which the local variable `<u_n>` is calculated. Note that it would not make sense to output this `<u_n>` separately over `<all cells>`, as the `<facenorm>` would be undefined.
- `ON <region>` *(optional)*: The output region for a local variable is only really used right now to specify what elements are output (should the `output` option be set).

Examples:

```
CELL_LOCAL <u_n> "dot(<u[l=:]>,cellave[lastface](<facenorm[l
   =:]>))"
CELL_LOCAL <d u_n d x[l=1]> "cellgrad[l=1](<u_n>)"
CELL_LOCAL <d u_n d x[l=2]> "cellgrad[l=2](<u_n>)"
CELL_LOCAL <d u_n d x_n> "dot(<d u_n d x[l=:]>,cellave[
   lastface](<facenorm[l=:]>))"
FACE_LOCAL <d^2 u_n d x_n^2> "facegrad(<d u_n d x_n>)" ON <
   boundaries> output
```

4.3.11 System variables

TODO

Search ref: system variables in setup_equations.pl

4.4 Simulation options

- TRANSIENT SIMULATION|STEADYSTATE SIMULATION: choose between the two types of simulation.
- END: end input
- START_SKIP|STOP_SKIP: ignore the text between these statements.
- DEFAULT_OPTIONS: add the following options to every subsequent variable, until cleared again using a blank DEFAULT_OPTIONS statement. When listed in order, default options precede a variable's individually specified options - hence, in the case of conflicting option statements, individual options take precedence over default options (ie, the individual options have a higher priority).
- OVERRIDE_OPTIONS: are the same as DEFAULT_OPTIONS, except that they follow a variable's individually specified options, and so in the case of conflicting option statements, take precedence over the individual options (ie, the override options have a higher priority).
- LINEAR_SOLVER: choose the type of linear solver to use.

4.5 Include statements and string replacements

Include statements allow other .arb input files to be included. These files can be user written, or be from a library of template files within the templates directory. String substitution that occurs as the file is read in allow these included files to be (basically) used as functions. An 'unwrapped' input file that is a handy reference as to how the include statements behaved (and can be used as an input file for subsequent runs) is placed at tmp/setup/unwrapped_input.arb after every run setup. The include statements are:

- INCLUDE_ROOT: choose a template directory to look for any files included via any following INCLUDE statements. If no string is specified, then the include root directory name is set equal to that of its parent (including) file. If the blank string is specified, then INCLUDE_ROOT is set to the blank string and all templates directories (up to two subdirectory levels) are searched for the following included files. The INCLUDE_ROOT definitions are hierarchical, in that the definition in a child file does not affect that of the parent.

Search ref: include root in setup_equations.pl

- INCLUDE: command to include a file from the most recent INCLUDE_ROOT directory (or subdirectory thereof), possibly also specifying file-specific string replacements using the syntax REPLACE "a string" WITH "another string" (or the shorter R "a string" W "another string"). If an INCLUDE_ROOT directory has not been specified (or cancelled with a blank INCLUDE_ROOT statement) then the templates directories will be searched until a matching INCLUDE file is found (up to two subdirectory levels right now).

Search ref: include in setup_equations.pl

- INCLUDE_WORKING: include the following files from the working directory. This command does not affect and is not influenced by the INCLUDE_ROOT directory and is (basically) used to include sets of user-written statements (i.e., like a local a function).

Search ref: include working in setup_equations.pl

Partnering the include file capability is the ability to read in multiple definitions for the same variable. The ultimate position of a variable's definition is that of the first definition for that variable. The ultimate expression used for a variable is that given (read in) last. This functionality allows a variable's expression to be changed from what is used in (say) a template file by specifying a new definition lower in the file, after the template file include statement. Options can also be added to previously specified options for a variable by including more definition statements (that may only contain options and not expressions) lower in the input file. Similarly for units.

There are two types of string replacements that occur when a line from an input file is parsed: i) file-specific replacements, which occur recursively through 'child' file inclusions, and ii) general replacements, which occur throughout all files from their point of definition onwards, until (possibly) cancelled. The following demonstrates a general replacement statement specifying two general replacement strings, using a long and short form:

Search ref: general replacements in setup_equations.pl

```
GENERAL_REPLACEMENTS REPLACE "<a region>" WITH "<another region>" R "a
string" W "another string"
```

There are certain system generated general replacements that occur automatically unless specifically changed by the user. Use the search hint to find the list of these in setup_equations.pl.

```
GENERAL_REPLACEMENTS CANCEL "<a region>" C "a string"
```

The above demonstrates how to cancel a search string replacement, using either a long (CANCEL) or short (C) form. Note that both general and file specific replacements do not occur on a line of an input file if the line is itself a general replacement definition line (specifically, it begins with the GENERAL_REPLACEMENTS keyword), or is an include line for a file (begins with some type of INCLUDE keyword).

4.6 Kernel options

There are many options that can be used to change the kernels used. For example

Search ref: kernel options
in kernel_module.f90

```
KERNEL polynomialaverageorder=2,polynomialorder=2
```

specifies that when averaging/differentiating quantities to/at faces, ensure that a second order polynomial would be reproduced precisely.

4.7 Glued boundaries

Used to implement periodic or reflection boundaries by glueing two boundary face regions together. Boundary regions to be glued must have the same element structure (size and number). Individual element matching between the boundaries is accomplished by matching the closest element locations, relative to the region centroids (much like the facelink and celllink operators).

Example of a periodic boundary glueing the top and bottom boundaries of a domain:

```
GLUE_FACES <south> <north>
```

Example of a reflection (axis of symmetry) boundary along the left side of a domain:

```
GLUE_FACES <west> reflect=1
```

In the case of reflection, certain operators (eg, facegrad) need to be aware when they are operating on the component of a vector, that needs to be reflected over this reflection boundary. See the reflect=1 options for each operator.

4.8 Simulation Info

The following strings can be used within an input file to help keep track of what the file contains. These and other automatically generated info strings are included as comments in most of the output files.

```
INFO_TITLE, INFO_DESCRIPTION, INFO_AUTHOR, INFO_DATE, INFO_VERSION
```

5 Expression language reference

There's lots missing in this section. The examples files are currently the best guide as to the language syntax.

5.1 Operators

5.1.1 General Notes

Operators produce a single value from the arguments that are contained within their parentheses (). They also accept options, contained within square brackets [].

```
operator[option1,option2,...](<argument1>,<argument2>,...)
```

The centring of an operator generally corresponds to the first syllable of the operator, however there are exceptions. Following the rule is `celldiv`, which is the cell centred divergence of a face centred quantity. This operator is cell centred and must be used in this context, hence its context centring is cell. The content expression passed into `celldiv` (actually its first argument) is face centred however. Similarly, `facegrad` is the gradient of a cell centred quantity evaluated at a face, so this operator is face (context) centred, but has cell content centring. Exceptions to the rule include the loop-type operators, `max`, `min`, `sum` and `product`. For example, `cellmax` loops through a region of cells finding the maximum value of an expression within those cells. Hence, this operator produces a result which has no centring (none centred) so can be used in any centring context, but has cell content centring.

Each operator accepts a certain number of arguments, however if an argument is not specified then a default value may be used. For example, `cellmax` uses three arguments: an expression that is to evaluated in each cell (`<expression>`, here denoted by a single variable, but more usually an expression of variables), an initial, default expression for the operator (`<default>`), and the cell centred content region over which the maximum will be calculated (`<region>`). Using implicit argument notation, operators expect the arguments in a specific order, so `cellmax` expects these three arguments in the manner

```
cellmax(<expression>,<default>,<region>)
```

If less than the required number of arguments are passed to an operator, then a default value for the omitted arguments will be assumed (or if no defaults are available or are sensible, an error will be flagged). For example, using

```
cellmax(<expression>)
```

sets `<default>` to −`<huge>` (the largest negative double precision number that the processor can store) and `<region>` to `<noloop>` if (for example) the expression was being used in a cell centred context. If in doubt about what the default value for an argument is, specify it!

The alternative to the implicit argument notation is to specify the arguments explicitly (similar to argument passing in f90). Using explicit notation the order of the arguments that are passed explicitly is irrelevant, however the order of any arguments that are not explicitly named (and hence specified implicitly) still is. For example, the following will all produce the same result

```
cellmax(expression=<expression>,default=<default>,region=<region>)
cellmax(<expression>,<default>,<region>)
cellmax(region=<region>,default=<default>,expression=<expression>)
cellmax(expression,region=<region>,default)
cellmax(region=<region>,expression,default)
```

Note in the last case that although `<expression>` was the second argument in the operator, it was the first implicitly named operator, so would be read correctly. Using a combination of the implicit and explicit passing is often convenient. For example, for the `cellmax` operator, the following form that uses a default value of `-<huge>` but performs the maximum comparison over a specified region is handy

```
cellmax(<expression>,region=<region>)
```

Operator options are similar to variable options. Some operators require a dimension, and this dimension (direction) is specified via the options. For example, `celldivgrad` calculates a gradient in a certain direction dimension using the divergence of a face centred scalar. To find this gradient in the second dimension you use the option `[l=2]`:

```
celldivgrad[l=2](<face centred expression>)
```

Some options are quite generic (eg, `noderivative`), however most are specific to the operator. There is no restriction on the order that options are specified.

Options and operators should be written in lowercase (I have started to make both of these case independent, but no guarantees yet).

Details of individual operators follows. Ultimate details of each operator (including argument order, options etc) can be found in the code file `src/setup_equations.pl` which shows how they are expanded into working code. Use search strings such as `ref: celldiv` within the perl file to find the specific code.

5.1.2 `celldiv`: Divergence

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

Statement:

```
celldiv[options](expression=<expression>)
```

Centring:

Operator is context cell centred, while `<expression>` is face centred.

Search ref: celldiv
in `setup_equations.pl`

Details:

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

Specifically, to discretise the divergence of a face centred vector $\boldsymbol{u}_\mathrm{j}$ over a cell i that sits within the domain, Gauss' theorem gives

$$\begin{aligned}\frac{1}{V_i}\int_{V_i}\boldsymbol{\nabla}\cdot\boldsymbol{u}dV &\Rightarrow \frac{1}{V_i}\sum_{j\in\mathbb{J}_{\text{nobcellfaces},i}}\frac{1}{S_j}\int_{S_j}\boldsymbol{N}_{i,j}\cdot\boldsymbol{u}_j\,dS \\ &= \underbrace{\sum_{j\in\mathbb{J}_{\text{nobcellfaces},i}}\frac{\boldsymbol{N}_{i,j}\cdot\boldsymbol{n}_j}{V_i}}_{\texttt{celldiv}}\frac{1}{S_j}\int_{S_j}\boldsymbol{n}_j\cdot\boldsymbol{u}_j\,dS \\ &\Rightarrow \texttt{celldiv(dot(<u[l=:]>,<facenorm[l=:]>))}\end{aligned}$$

where V_i and S_j are the volume and total surface area of the cell i and face j, respectively, $\boldsymbol{N}_{i,j}$ is a unit normal pointing outward from cell i but located at face j, $\boldsymbol{n}_j$ is a normal associated with face j, and the sum is conducted over the set of all face elements that surround cell i, denoted by $\mathbb{J}_{\text{nobcellfaces},i}$. In the equivalent coding the face centred vector $\boldsymbol{u}_j$ 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 as the divergence of a vector results in a scalar, the above operation produces a scalar for each cell it is performed in.

The region used by arb in performing the above sum as represented by $\mathbb{J}_{\text{nobcellfaces},i}$ is `<nobcelljfaces>` ('no-boundary-cell-faces'). 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
```

5.1.3 `cellgrad` or `facegrad`: Gradient

Search ref: cellgrad, ref: facegrad
in `setup_equations.pl`

Summary: Calculates a scalar component of a gradient over a cell or face.

Statement:

```
cellgrad[options](expression=<expression>)
facegrad[options](expression=<expression>)
```

Centring:

`cellgrad` is context cell centred and `facegrad` is context face centred. In both cases `<expression>` is cell centred.

Details:

To calculate the gradient of a cell centred scalar ϕ_i in coodinate direction 2 in cell i,

$$\frac{1}{V_i}\int_{V_i} \boldsymbol{e}_2 \cdot \boldsymbol{\nabla}\phi dV \Rightarrow \sum_{i' \in \mathbb{I}_{\text{cellcells},i}} \overset{\bullet(2)}{k}_{i,i'}\phi_{i'} \Rightarrow \texttt{cellgrad[l=2](phi)}$$

where $\boldsymbol{e}_2$ is a unit vector in coordinate direction 2, $\overset{\bullet(2)}{k}_{i,i'}$ is a predetermined kernel for this operation, and $\mathbb{I}_{\text{cellcells},i}$ is the set of all cells in the vicinity of cell i that are used by this kernel. Kernels to calculate the cell gradient in the other coordinate directions, that is $\overset{\bullet(1)}{k}_{i,i'}$ and $\overset{\bullet(3)}{k}_{i,i'}$ also exist.

A gradient of a cell centred quantity evaluated at a face can be calculated similarly, for example

$$\frac{1}{S_j}\int_{S_j} \boldsymbol{e}_3 \cdot \boldsymbol{\nabla}\phi dS \Rightarrow \sum_{i \in \mathbb{I}_{\text{facecells},j}} \overset{\circ(3)}{k}_{j,i}\phi_i \Rightarrow \texttt{facegrad[l=3](phi)}$$

Gradients taken in directions relative to the face orientation are also available using the `facegrad` operator. Index 4 gives the gradient relative to the face's normal, that is

$$\frac{1}{S_j}\int_{S_j} \boldsymbol{n}_j \cdot \boldsymbol{\nabla}\phi dS \Rightarrow \sum_{i \in \mathbb{I}_{\text{facecells},j}} \overset{\circ(4)}{k}_{j,i}\phi_i \Rightarrow \texttt{facegrad[l=4](phi)}$$

In computational terms the face normal is represented by (`<facenorm[l=1]>`,`<facenorm[l=2]>`, `<facenorm[l=3]>`). Indices 5 and 6 give gradients in the directions of the first and second tangents for each face, respectively, that is

$$\frac{1}{S_j}\int_{S_j} \boldsymbol{t}_j^{(1)} \cdot \boldsymbol{\nabla}\phi dS \Rightarrow \sum_{i \in \mathbb{I}_{\text{facecells},j}} \overset{\circ(5)}{k}_{j,i}\phi_i \Rightarrow \texttt{facegrad[l=5](phi)}$$

and

$$\frac{1}{S_j}\int_{S_j} \boldsymbol{t}_j^{(2)} \cdot \boldsymbol{\nabla}\phi dS \Rightarrow \sum_{i \in \mathbb{I}_{\text{facecells},j}} \overset{\circ(6)}{k}_{j,i}\phi_i \Rightarrow \texttt{facegrad[l=6](phi)}$$

Computationally $\boldsymbol{t}_j^{(1)}$ is represented by (`<facetang1[l=1]>`,`<facetang1[l=2]>`, `<facetang1[l=3]>`) and $\boldsymbol{t}_j^{(2)}$ by (`<facetang2[l=1]>`,`<facetang2[l=2]>`, `<facetang2[l=3]>`), respectively. If the face has one dimension then $\boldsymbol{t}_j^{(1)}$ will be directed along the face, and $\boldsymbol{t}_j^{(2)}$ will be normal to both $\boldsymbol{t}_j^{(1)}$ and $vecti[j]n$. If the face has no or two dimensions (a point or a plane) then there are no preferential directions for these tangents. If no index is specified on the `facegrad` operator then `l=4` is assumed.

Options:

- `l=1`, `l=2`, etc: This index specifies the direction that the gradient will be taken in. For `cellgrad` this index represents the dimension the gradient is taken in and must be specified. For `facegrad` if the index is specified and is ≤ 3, this specifies the dimension the gradient is taken in. For an index ≥ 4, the direction is taken relative to the face orientation. `l=4` specifies a gradient taken in the direction of the face normal, `l=5` a gradient taken in the direction of the first tangent to the face and `l=6` in the direction of the second tangent to the face. If the index is not specified for `facegrad` then `l=4` is assumed — that is, a gradient taken normal to the face.

- `adjacentcells` for `facegrad` only: Gradient is based on adjacent cells only, but attempts to be in the direction of the face normal (it is only an approximation, but should be accurate for structured meshes). Note, only works for `l=4` direction — that is, the direction of the face normal.

- `dxunit` for `facegrad` only: Similar to option `adjacentcells` in that it is based on adjacent cells only, but now it is in the direction of `<dxunit[l=:]>`, which is a unit vector pointing from the centre of the cell immediate below the face (in the face's normal direction) to the centre of the cell immediately above the face. Hence, for unstructured meshes, this gradient is not precisely in the same direction as the true `<facenorm[l=:]>`.

- `reflect=1`, `reflect=2`, etc: This specifies that the contained expression is a component of a vector, and that over any glued reflection boundaries, must be reflected in this direction. These options only need to be specified if the operator is going to be acting over (or next to) a glued, reflection boundary that is reflected in a direction that is the same as the vector's component direction.

- `noderivative`: As previously.

Examples:

```
FACE_DERIVED <T flux> "-<D>*facegrad(<T>)" ON <all faces> #
   some type of heat flux occuring across each face
CELL_DERIVED <dpdx[l=1]> "cellgrad[l=1](<p>)" # gradient of
   pressure in first dimension
```

5.1.4 `cellave`: Interpolation to cell centring

Search ref: cellave
in `setup_equations.pl`

Summary: Interpolates or averages an expression from (mainly) face centring to cell centring.

Statement:

`cellave[options](expression=<expression>)`
Centring:

`cellave` is context cell centred and generally takes a face centred expression (see `othercell` option however).

Details:

Without any options, predefined kernels are used to interpolate the face centred expression from the faces that surround a cell to the centroid of that cell.

Options:

- `lastface`: Evaluates `<expression>` at the last face that was referenced in the context of the operator's position, but treats the result as having cell centring.
- `lastfacenoglue`: As above, but moves through glued boundaries to the actual last face that was used (if it was glued).
- `othercell`: Evaluates `<expression>` at the cell that is adjacent to the last face that was referenced in the context of the operator's position. In this (exception) case `<expression>` is cell centred. For this case only `reflect=1` etc options may be used/necessary as the cell may be on the other side of a glued reflection boundary.
- `noderivative`: As previously.

5.1.5 `faceave`: Interpolation

Search ref: faceave
in `setup_equations.pl`

Summary: Interpolates or averages an expression from cell to face centring.

Statement:

`faceave[options](expression=<expression>)`

`faceave[advection,options](expression=<expression>,flux=<flux>,`
`limiter=<limiter>)`

Centring:

`faceave` has face context centring. `<expression>` is cell centred. `<flux>` is face centred. `<limiter>` is cell centred.

Details:

TODO

Options:

- `harmonic`:
- `advection`:
- `lastcell`:
- `noderivative`: As previously.

5.1.6 `cellsum` or `facesum`: Sum

Summary: Performs a sum over a region of either cell or face elements.

Statement:

```
cellsum[options](cell_centred_expression)
cellsum[options](cell_centred_expression,<cell_centred_region>)
facesum[options](face_centred_expression)
facesum[options](face_centred_expression,<face_centred_region>)
```

Centring:

Operators may be cell, face or none centred. Contents of `cellsum` is cell centred, contents of `facesum` is face centred.

Details:

This operator sums the contained expression over a region of cell or face elements. If no region is specified, then default regions are applied, defined by:

Operator centring	Expression centring	Default region
`cell`	`face`	`<celljfaces>`
`cell`	`cell`	`<adjacentcellicells>`
`face`	`cell`	`<adjacentfaceicells>`
all else		`<noloop>`

Options:

- `noderivative`: As previously.

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

5.1.8 `celllimiter`: Gradient limiter for ensuring advection stability

5.1.9 `cellif`, `faceif` or `noneif`: If conditional statement

5.1.10 `cellproduct` or `faceproduct`: Product performed over a region of elements

5.1.11 `cellmin/max`, `facemin/max` or `nonemin/max`: Picks the minimum/maximum from a region of elements

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

5.1.13 `celllink` or `facelink`: Link to other regions

6 Code structure